华中科技大学

课程设计报告

题目：基于 SAT 的双数独游戏求解程序

课程名称：程序设计综合课程设计

专业班级：大数据 2101 班

学　　号：U202115652

姓　　名：李嘉鹏

指导教师：李开

报告日期：2022 年 9 月 8 日

计算机科学与技术学院

任务书

□ 设计内容

SAT 问题即命题逻辑公式的可满足性问题（satisfiability problem），是计算机科学与人工智能基本问题，是一个典型的 NP 完全问题，可广泛应用于许多实际问题如硬件设计、安全协议验证等，具有重要理论意义与应用价值。本设计要求基于 DPLL 算法实现一个完备 SAT 求解器，对输入的 CNF 范式算例文件，解析并建立其内部表示；精心设计问题中变元、文字、子句、公式等有效的物理存储结构以及一定的分支变元处理策略，使求解器具有优化的执行性能；对一定规模的算例能有效求解，输出与文件保存求解结果，统计求解时间。在得到 DPLL 求解器后，还要将其与实际的应用问题相结合，例如将双数独游戏格局规约为 CNF 子句集，并进一步实现对游戏的求解和生成功能。

□ 设计要求

要求具有如下功能：

（1）输入输出功能：包括程序执行参数的输入，SAT 算例 cnf 文件的读取，执行结果的输出与文件保存等。(15%)

（2）公式解析与验证：读取 cnf 算例文件，解析文件，基于一定的物理结构，建立公式的内部表示；并实现对解析正确性的验证功能，即遍历内部结构逐行输出与显示每个子句，与输入算例对比可人工判断解析功能的正确性。数据结构的设计可参考文献[1-3]。(15%)

（3）DPLL 过程：基于 DPLL 算法框架，实现 SAT 算例的求解。(35%)

（4）时间性能的测量：基于相应的时间处理函数（参考 time.h），记录 DPLL 过程执行时间（以毫秒为单位），并作为输出信息的一部分。(5%)

（5）程序优化：对基本 DPLL 的实现进行存储结构、分支变元选取策略[1-3]等某一方面进行优化设计与实现，提供较明确的性能优化率结果。优化率的计算公式为：$[(t-t_o)/t]*100\%$,其中 t 为未对 DPLL 优化时求解基准算例的执行时间，t_o 则为优化 DPLL 实现时求解同一算例的执行时间。(15%)

（6）SAT 应用：将数双独游戏[5]问题转化为 SAT 问题[6-8]，并集成到上面

的求解器进行数独游戏求解，游戏可玩，具有一定的/简单的交互性。应用问题归约为 SAT 问题的具体方法可参考文献[3]与[6-8]。(15%)

□ 参考文献

[1] 张健著. 逻辑公式的可满足性判定—方法、工具及应用. 科学出版社，2000

[2] Tanbir Ahmed. An Implementation of the DPLL Algorithm. Master thesis, Concordia University,Canada,2009

[3] 陈稳. 基于 DPLL 的 SAT 算法的研究与应用.硕士学位论文，电子科技大学，2011

[4] Carsten Sinz. Visualizing SAT Instances and Runs of the DPLL Algorithm. J Autom Reasoning (2007) 39:219–243

[5] 360 百科：数独游戏 https://baike.so.com/doc/3390505-3569059.html

Twodoku： https://en.grandgames.net/multisudoku/twodoku

[6] Tjark Weber. A sat-based sudoku solver. In 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2005, pages 11–15, 2005.

[7] Ins Lynce and Jol Ouaknine. Sudoku as a sat problem. In Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics, AIMATH 2006, Fort Lauderdale. Springer, 2006.

[8] Uwe Pfeiffer, Tomas Karnagel and Guido Scheffler. A Sudoku-Solver for Large Puzzles using SAT. LPAR-17-short (EPiC Series, vol. 13), 52–57

[9] Sudoku Puzzles Generating: from Easy to Evil.

http://zhangroup.aporc.org/images/files/Paper_3485.pdf

[10] 薛源海，蒋彪彬，李永卓. 基于“挖洞”思想的数独游戏生成算法. 数学的实践与认识,2009,39(21):1-7

[11] 黄祖贤. 数独游戏的问题生成及求解算法优化. 安徽工业大学学报(自然科学版), 2015,32(2):187-191

目录

1 引言

1.1 课题背景与意义

本课程设计以 SAT 问题的求解作为背景，以双数独游戏作为问题的载体，使我认识到了一种在国际上具有悠久历史的经典难题，同时加深了我对应用问题与理论问题相互转换的了解。

由于 SAT 问题是被首先证明的 NP 完全问题，它可以被视作所有 NP 完全问题的“种子”，因而所有 NP 问题都可以在多项式时间内被转化为 SAT 问题并进行求解。正是由于 SAT 问题在相关领域研究中具有“举一反三”的广泛应用，它也始终是计算机理论研究中的热点问题。

1.1.1 SAT 问题简介

SAT 问题就是命题逻辑公式的可满足问题（satisfiability problem）。它作为一个计算机科学与人工智能领域的基本问题，可以在多个领域（如 AI 开发中的自动驾驶模型、实际生产生活中的调配方案、EDA 领域中芯片布局布线设计环节的优化、网络安全协议的认证等）起到至关重要的作用，具有重要理论意义和应用价值。

对于一个给定的 SAT 问题，它包含一定量的布尔变元（Bool）集合（又被称为文字）：

$$\{x_1, x_2, \ldots, x_n\}$$

以及由这些布尔变元集合构成的一定量的析取范式（也就是一个或多个布尔变元的“或”运算，也被称为“子句”）：

$$c_{m0}= x_i \vee x_j \vee \ldots \vee x_k \quad (1 \leqslant i, j, k, \ldots \leqslant n)$$

SAT 问题的求解过程实质上是对一个合取范式（即 CNF 公式）

$$F= c_1 \wedge c_2 \wedge \cdots \wedge c_m$$（其中可能有空子句）

判定是否存在一组对全部布尔变元的赋值，使得在该赋值条件下 CNF 公式的结果为真。

1.1.2 DPLL 算法简介

为了高效率求解 SAT 问题，我采用了目前被广泛使用的 DPLL 算法（即基于二叉树的回溯搜索算法）。DPLL 算法主要运用两种基本规则对 CNF 子句集

进行化简：

其一是单子句规则，顾名思义，此规则需要在子句集中找到单子句。若可以找到，则该单子句一定为真，于是可以从公式中删除所有包含该单子句文字的子句（析取范式一定为真）。若此时子句集为空，说明 CNF 公式可满足；否则遍历余下的子句，从包含“该单子句文字的负文字”的子句中删除它，于是原公式简化为了现在的公式。

其二是分裂策略。在上述过程结束后，子句集中不存在单子句，因此为了使算法进行下去，可以选取一个文字，将其作为新的单子句加入 CNF 公式，并重复以上过程。

不断交互使用以上两条规则，最终可在某一步得到 CNF 公式的可满足性。当然，分裂策略中新变元的选取方式会极大地改变 DPLL 算法的运行性能，因此优化的主要突破口就是选择更合适的新变元。

1.1.3 双数独游戏与 SAT 问题的相互转换与联系

双数独游戏是目前风靡全球的益智游戏之一，它可以通过转化为 SAT 问题来求解，具体体现在以下多条原则中：每个单元格只能填入唯一一个数字、每行（和每列）都要包含 1~9 中的每个数字且各个数字只出现一次、每个小盒子都要包含 1-9 中的每个数字且各个数字只出现一次。同时，还要考虑中心的 9 个格子由两个单独的数独共享。由此可以根据具体条件设置不同的子句，全部子句合在一起就构成了双数独游戏的全部 CNF 公式。

1.2 国内外研究现状

近十年来，国内外对 SAT 问题求解的研究已经取得了很大突破，在各大 SAT 算法竞赛中也能看到各国团队的身影。目前，SAT 问题中较经典的求解算法有 DPLL（Davis Putnam Logemann Loveland）算法，CDCL（Conflict Driven Clause Learning）冲突驱动子句学习算法等[1]。除了常规的回溯、剪枝等策略，现有 SAT 求解算法中还有许多关键技术尚待进一步研究，如变元选择决策（这是 DPLL 算法中改进的重要因素）、预处理、非时序回溯等。不过，近二十年来，各式各样的求解器的问世使得可求解 SAT 问题的规模不断增大，已经基本可以完全满足相关领域内的绝大部分需求。人们也提出了许多新的理论和方向，以更加深入地探索 SAT 问题及其应用。

1.3 课程设计的主要研究工作

进行本次课设任务之前，我首先上网查阅了 SAT 问题和 DPLL 算法的基础知识，结合课程设计任务书对整体流程有了基本了解。此后，我首先成功解析了.cnf 公式文件并采用初级的 DPLL 算法对相应 CNF 公式进行求解。接下来，我通过查阅资料、数学推理和与同学交流等多种方式，确定了我的优化策略，并进一步将双数独游戏转化为 CNF 公式进行求解，最终实现了课程设计的全部内容。

2 系统需求分析与总体设计

2.1 系统需求分析

根据 2021 级课程设计任务书，我的系统需要满足两个方面的需求，详细功能如下：

第一部分是 SAT 问题求解模块，可以对 CNF 公式进行一系列操作，包括：

① 读取（输入）、输出、删除、解析公式。这四项主要是对 CNF 公式本身的操作，是程序的基础；（这里在使用相应功能时需要进行判空操作，即判断 CNF 文件是否存在，否则可能会引发访问越界等异常）

② DPLL 求解、测量所用时间、算法策略优化等。这是本次课设的核心元素。

第二部分是双数独游戏模块，可以对一个给定的未完成双数独游戏求解，也可以逆向生成一个双数独游戏（同时也包含与用户的实时交互，如对填写结果正确与否的反馈）。此模块中需要对用户输入的数值进行合法性检测（例如生成数独时的挖洞个数、用户游玩游戏时的输入数值等）。

2.2 系统总体设计

根据 2.1 节中所涉及到的功能需求，系统分为 SAT 问题求解和双数独游戏两大模块。在程序刚开始运行时，首先需要用户在初始菜单中选择使用哪个模块（如图 2-1 所示）。

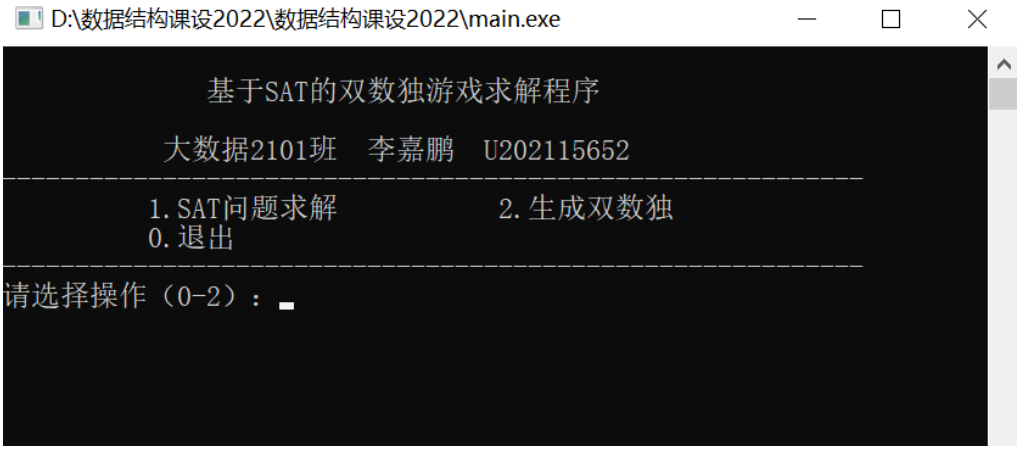

图 2-1 初始菜单

对于 SAT 问题求解模块，首先需要给出菜单画面（如图 2-2 所示），菜单选项包括读取 CNF 文件、遍历并输出子句、DPLL 求解（同时输出到.res 文件）、去除当前子句集和退出五个选项，用户输入想要使用的功能数字后，可以完成相应任务。

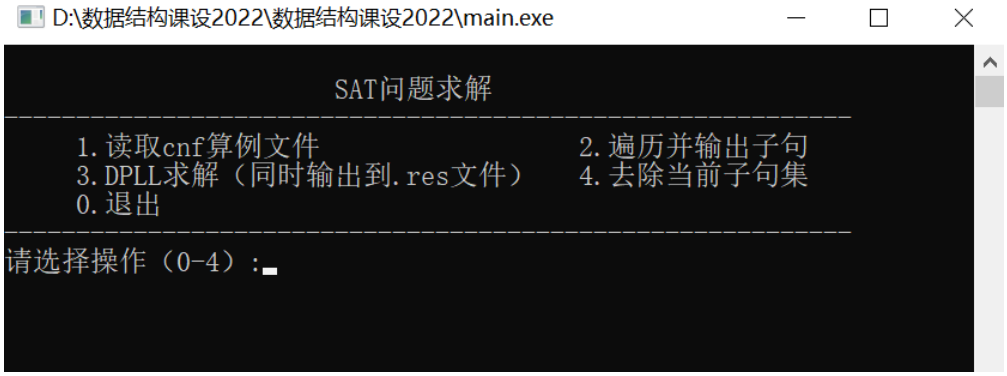

图 2-2 SAT 求解模块菜单

对于双数独游戏模块，首先同样要给出菜单选择画面，选项包括求解双数独游戏、生成双数独和退出三个选项。

综上所述，完整的系统总体结构图如图 2-3 所示。

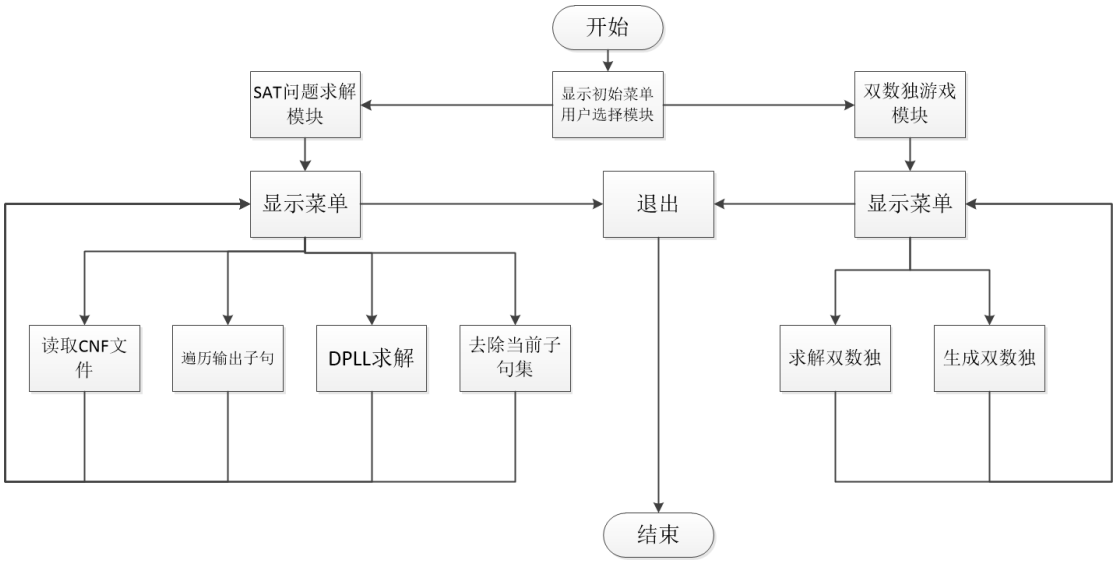

图 2-3 系统总体结构图

3 系统详细设计

3.1 有关数据结构的定义

本系统主要处理的数据类型有变元（文字）、子句、CNF公式等，其包含关系由左至右。由于公式中变元和子句数可能不同，且同一条子句中含有的变元数也大都不等（甚至有时候相差悬殊），为了避免大量空间浪费和查找的繁琐，不便用传统的数组来存储。因此我采用了二重链表表示法，即子句是由文字构成的链表，公式又是由子句构成的链表。这样的数据结构既可以支持DPLL算法中的回溯过程，又在删除、插入等多个方面具有相当大的灵活性。

对于文字，其包含两个数据项，分别是文字的值（正值代表正文字，反之为负文字）和文字的next指针域（指向同一子句内的下一个文字或NULL）。

对于子句，其包含三个数据项，分别是子句内的文字总数（在后续查找单子句等函数中起作用）、子句的head指针域（指向该子句的第一个文字，若为空子句则为NULL）和子句的next指针域（指向该子句的下一个子句）。

对于公式，它需要存储一个CNF文件中的全部信息，还要记录在DPLL求解过程中的一些变化，因此由较多数据项组成，包括：整个公式的文字总数、子句总数、在整个过程中的空子句总数、单子句总数和当前子句总数，以及指向整个公式第一个子句的head指针域。

以上三种数据类型及其包含的数据项可用表3-1表示。

表 3-1 文字、子句、公式的数据结构

数据类型	包含的数据项
文字	文字的值（word） next 指针域
子句	子句内的文字总数（num） head 指针域 next 指针域
公式	公式的文字总数（clausenum） 公式的子句总数（literalnum） 公式的空子句总数（emptyclausenum） 公式的单子句总数（singleclausenum） 公式的当前子句总数（currentclausenum） head 指针域

图3-1直观地表示了这三种数据类型在系统中如何关联。

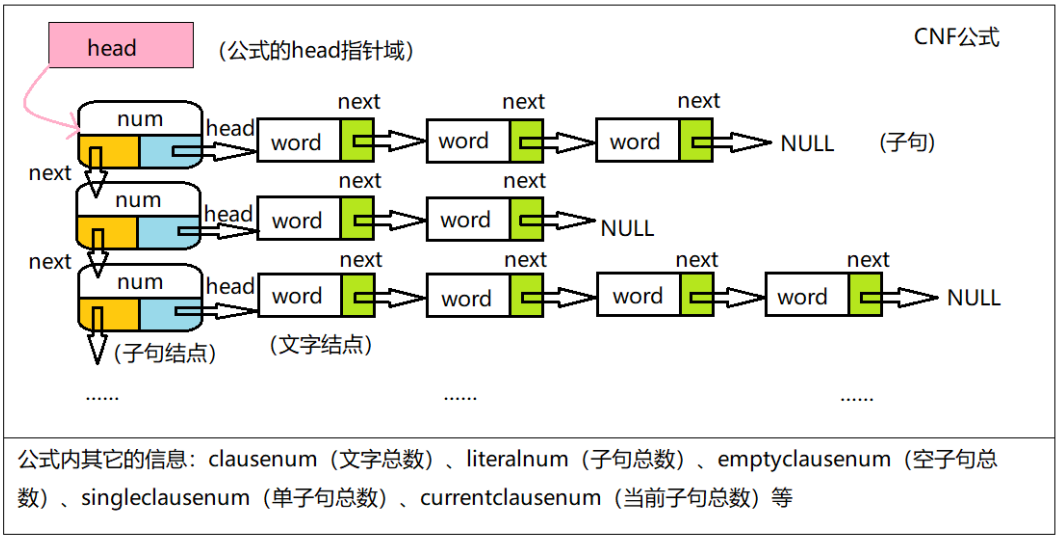

图3-1 文字、子句、公式的系统关联

3.2 主要算法设计

本系统的SAT问题求解模块和双数独游戏求解模块都由模块化的函数和算法提供支撑。SAT问题求解模块首先需要读取CNF算例文件。为此需要使用文件指针对相应文件进行读操作，同时边读入边进行链表结点的创建（此处需要特别注意避免链表指针为野指针），全部读入完成后将相关信息写入公式的数据项。DPLL算法思想和原理已在前文进行描述，此处为了体现出改进前后的性能优劣，在开始求解前可以选择策略（改进前和改进后）。对于DPLL算法，首先依据公式内的单子句数量判断是否存在单子句，如果存在则开始搜索单子句并依据单子句规则进行两次删除操作并不断循环并判断，直到无法找到单子句后再根据所选策略选取新变元并作为单子句加入原子句集，然后分别开始DPLL在加入新变元和新的负变元的递归过程。若要更换算例，需要销毁链表、释放相应空间并把公式的head指针域置为NULL。

双数独游戏模块中，若要对一个未完成的双数独进行求解，首先需要根据双数独游戏的基本规则生成基本的CNF规约条件，然后将给定数独格局中的提示数信息一一转化为对应的单子句，这样得到完整的CNF文件后使用DPLL算法进行求解得到答案，再将相应的变元进行转换填入双数独空缺位置即可。而从逆向的角度看，如果要随机生成一个双数独游戏，可以先从一个完整的双数独格局开始（利用上述求解功能很容易得到一个完整的初始合法棋盘格局），采用挖洞法思

想，在每一步随机确定好挖洞位置后，尝试填入其它8个数字是否仍能得到解。如果无解，说明挖去这个数后可以保证有唯一解，反之则无法挖洞，需要更换挖洞位置。此处可以让用户自行输入想要挖洞的数量，从而影响双数独游戏的难度。在用户游玩游戏的过程中，需要实时判断用户输入是否正确，若正确则给予反馈并在双数独棋盘中相应的位置补入正确答案，以达到一定的可玩性与交互性。

两个模块的算法思想和流程分别如图3-2和图3-3所示。

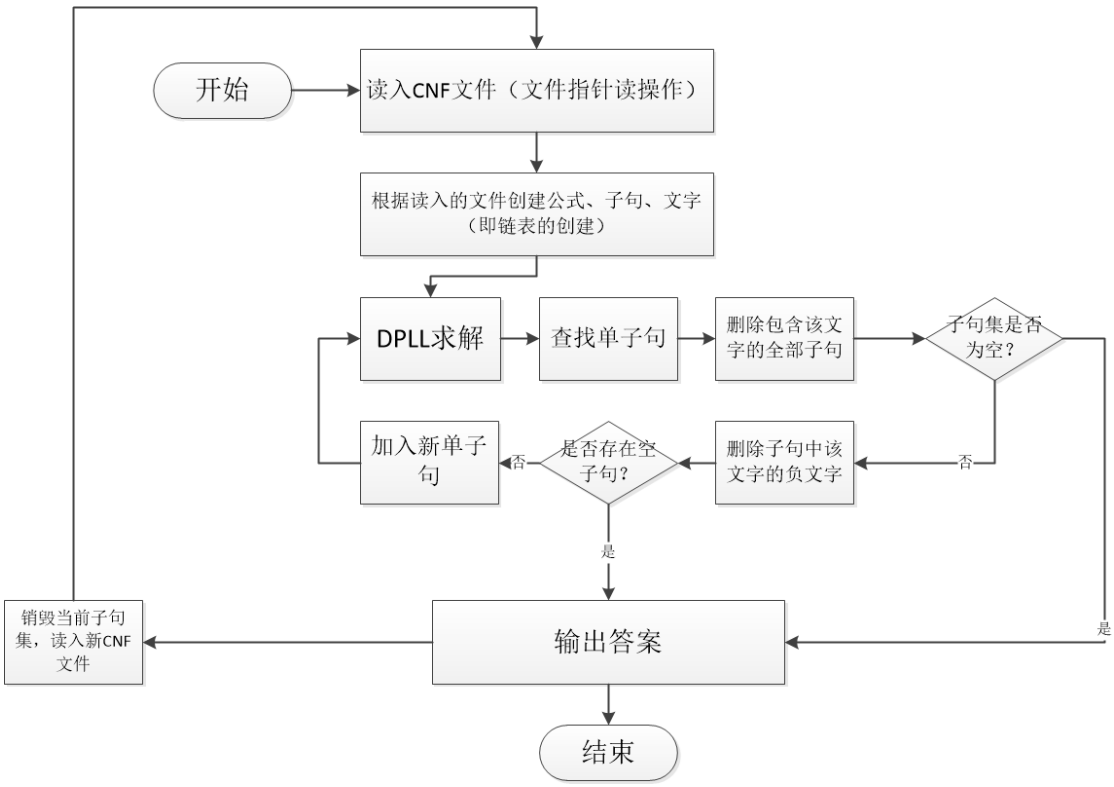

图3-2 SAT问题求解模块的算法流程图

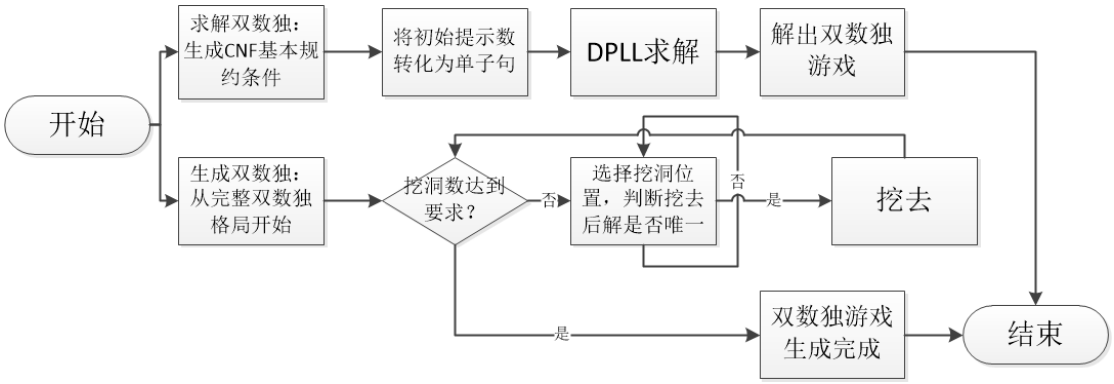

图3-3 双数独游戏模块的算法流程图

4 系统实现与测试

4.1 系统实现

本系统在 Windows 10 操作系统环境下可以正常运行，代码通过 Dev-C++编辑器编写、调试、编译。

4.1.1 数据结构的 C 语言定义

根据 3.1 节中有关数据结构的设计，用 C 语言定义了公式、子句和文字等数据类型，如下：

```
typedef struct literal { //以链表形式存储文字
    int word;
    struct literal *next;
} literal, *L;

typedef struct clause { //以链表形式存储子句
    int num; //某子句内的文字总数
    struct literal *head; //第一个文字
    struct clause *next; //下一个子句
} clause, *C;

typedef struct formula { //整个公式
    int clausenum;//文字总数
    int literalnum;//子句总数
    int emptyclausenum;//空子句总数
    int singleclausenum;//单子句总数（随 DPLL 进行而减少）
    int currentclausenum;//当前子句总数（随 DPLL 进行而减少）
    struct clause *head;//第一个子句
} formula, *F;
```

4.1.2 主要函数说明与分析

（一）SAT 问题求解模块函数与算法

（1）void cnfopen(formula &original, char *input)

函数功能：读入 CNF 文件，将其中的文字、子句等信息存储到二重链表中。

函数实现：使用文件指针 fp，首先读取到前面的注释和“p cnf”处，然后读取公式中的文字总数和子句总数，接下来对每一条子句依次读入文字信息，边读入边创建文字结点，从而完成对链表的建立。

复杂度分析：若有 a 个子句且每句平均含有 b 个文字，则读入的时间复杂度为 O(a*b)；读入每条子句时为便于一次性生成链表，开辟了临时存储文字的数组空间 O(1)，再加上开辟的链表空间 O(a*b)，总空间复杂度为 O(a*b)。

（2）void printformula(formula &original)

函数功能：遍历每条子句并输出。

函数实现：使用两个搜索指针对整个公式进行遍历，一个指针遍历子句，另一个指针从该子句的头文字开始，在当前子句内遍历所有文字，然后逐个输出。

复杂度分析：若有 a 个子句且每句平均含有 b 个文字，则时间复杂度为 O(a*b)，空间复杂度为 O(1)。

（3）void deleteformula(formula &original)

函数功能：删除整个 CNF 公式并释放全部空间。

函数实现：如（2）中所述，对整个公式进行遍历，同时用另外的两个指针记录当前搜索指针的后一位的位置（避免删除过程中前面结点被删除导致后面结点无法寻址），然后逐个释放结点的内存空间并将相应指针置为 NULL。

复杂度分析：时间复杂度为 O(a*b)，空间复杂度为 O(1)。

（4）int findunitclause(formula &original)

函数功能：查找公式内是否存在单子句，如果存在则返回一个单子句内文字的值，否则返回 0。

函数实现：首先判断公式的数据项 singleclausenum（单子句个数）是否为 0。如果为 0，则直接返回 0；否则对子句集进行遍历，如果找到一个仅含 1 个文字的子句则说明其为单子句，输出即可。

复杂度分析：若不存在单子句，时间复杂度为 O(1)；若存在单子句，遍历所需的时间复杂度为 O(a)。二者的空间复杂度为 O(1)。

（5）void destroyclause(formula &original, int unitclause)

函数功能：删除所有包含某文字的子句，对公式的相关数据项做出相应调整。

函数实现：对子句集进行遍历，若找到某子句内含有给定的文字则进行删除，并

将公式的 currentclausenum（当前子句数）减 1。

复杂度分析：要遍历整个公式，故时间复杂度为 O(a*b)，空间复杂度为 O(1)。

（6）void deleteclause(formula &original, int negative)

函数功能：去除全部子句中的负文字，对子句的相关数据项做出相应调整。

函数实现：依然是遍历整个子句集，若找到某子句内含有给定的负文字则将这个文字结点删除。此处如果发现原子句本身为单子句，则需要将公式的数据项 emptyclausenum（空子句个数）加 1。

复杂度分析：时间复杂度为 O(a*b)，空间复杂度为 O(1)。

（7）int existemptyclause(formula &original)

函数功能：判断公式内是否存在空子句，如果存在返回 1，否则返回 0。

函数实现：从头子句开始遍历，如果某一字句所含的文字数为 0 则说明存在空子句，遍历结束。

复杂度分析：时间复杂度为 O(a)，空间复杂度为 O(1)。

（8）int isemptyformula(formula &original)

函数功能：判断公式是否为空，若为空返回 1，否则返回 0。

函数实现：只需检验公式的 currentclausenum（当前子句数）是否为 0 即可。

复杂度分析：时间和空间复杂度均为 O(1)。

（9）void copyformula(formula &temp, formula &original)

函数功能：对公式进行复制。

函数实现：和读取函数的算法完全一致，特点是边读边写，即为新公式依次开辟二重链表空间，并将原公式的子句。文字、各种相关数据逐个写入。

复杂度分析：时间复杂度、空间复杂度均为 O(a*b)。

（10）void addunitclause(formula &original, int key)

函数功能：在原公式中添加单子句。

函数实现：如果原公式不存在，则将新单子句作为公式的头子句加入；否则，在原公式的头子句前添加新单子句即可（从而避免了遍历到子句集的末尾）。添加完成后，还要把公式的 currentclausenum（当前子句数）和 singleclausenum（单子句总数）分别加 1。

复杂度分析：时间复杂度、空间复杂度均为 O(1)。

（11）int DPLL(formula &original, int strategy)

函数功能：DPLL 核心求解算法，通过两种不同策略对原 SAT 问题进行求解。

函数实现：①单子句规则：

a. 首先查找原公式中是否有单子句；

b. 如果有，则认为该文字一定为真，并在全部子句集中删除包含该单子句的子句，判断此时公式是否为空；

c. 如果为空，则说明公式可满足，输出结果；否则在全部子句集中删除包含该单子句的子句，判断此时公式是否含有空子句。如果有，则说明公式不可满足，输出结果，否则重复 a-c 直到无法找到单子句。

②分裂策略：

a. 当①中无法找到单子句后，将原公式复制一份副本，便于后续递归操作；

b. 按照选取的策略选择新变元，并将新变元和新的负变元分别作为新的单子句加入两份公式。如果选择优化前的策略，则是按照从前到后的顺序（头子句的第一个文字、第二个文字……）选取新变元。这种方式具有一定的随机性，往往在面对大算例时表现出的性能不够出色。而优化后的策略则统计了正、负文字的出现次数，并选取出现次数最多的正（如果没有正文字，就是负）文字，这样可以使得加入新单子句后可删去的子句和“包含该负文字的子句”数量大大提高，有助于更快地简化公式，从而在极大程度上提高 DPLL 算法的求解速率。

c. 对以上得到的两个新公式分别再运用 DPLL 求解（递归）。

复杂度分析：设初始状态下公式的文字总数为 p，单子句个数为 q。首先，在查找单子句的过程中需要遍历子句集 q 次，找到后需要删除包含该文字的子句和子句中的相应负文字，最坏情况下的时间复杂度为 O(q*a+2a*b)，空间复杂度为 O(1)。对于第一种策略，选取新变元的时间复杂度为 O(1)，回溯过程的空间复杂度为 O(p)，总的空间复杂度为 O(p)；对于第二种策略，选取新变元需要遍历整个公式，时间复杂度为 O(a*b)，总的空间复杂度为 O(p)。

（12）void saveres(char *output, formula &original, int time, int DPLLstatus)

函数功能：将 DPLL 求解的结果输出到同名的.res 文件中。

函数实现：使用文件指针 fp 对目标文件进行写操作，首先需要对输入的 CNF 算

例文件名进行保存并将后三个字母改为“res”，然后写入求解结果（1 或 0），遍历结果数组写入文字的真值，最后写入求解时间（用两个计时器变量相减）。

复杂度分析：时间复杂度为 O(p)，空间复杂度为 O(1)。

（二）双数独游戏模块函数与算法

（1）int convert(int x)

函数功能：将四位数语义编码（指形如 aijd 的数独位置-数字信息，a 代表左上/右下大数独，i 和 j 分别代表格子的行、列位置，d 代表格子中填的数）从 1 到 1458 连续编码。

函数实现：由于 a 取值为 1 或 2，i、j、d 取值均为 1~9 之间的整数，因此设计转换公式，得到的结果等于（x 的个位+（x 的十位-1）*9+（x 的百位-1）*81+（x 的千位-1）*729），这样转换的结果就是从 1 开始的连续编号，符合 CNF 文件的变元编码规范。

复杂度分析：时间和空间复杂度都是 O(1)。

（2）int reverse(int x)

函数功能：将变元还原为语义编码。

函数实现：实际上就是上面转换函数的逆过程，分别分离个、十、百、千位即可。

复杂度分析：时间和空间复杂度都是 O(1)。

（3）void addclausefull(formula &newdouble, int count, int temp[50])

函数功能：在数独的 CNF 公式中添加完整的子句。

函数实现：使用一个 temp 数组来暂存子句中文字的信息，然后依次创建文字结点并填入数据。此处不直接边读入边创建的原因是后续进行双数独游戏 CNF 条件规约时需要多次使用循环，利用数组存变元更方便。

复杂度分析：设 c 为该子句中的文字数量，则时间复杂度和空间复杂度均为 O(c)。

（4）void generatecnf(formula &newdouble)

函数功能：生成双数独游戏 CNF 子句集。

函数实现：首先对双数独游戏的 CNF 公式进行初始化，然后加入提示数信息（一个提示数相当于一个单子句）。接下来分别依据四种格局规约加入约束条件子句：

① 每个单元格只能填入唯一一个数字（此处将中间重叠的 9 个格视为两个单独的格子）。对于每个格子，此约束会产生 37（=1+$C\frac{2}{9}$）条子句，因此共有 2

×9×9×37=5994 条子句。

②行约束：每行需要填入 1~9 中的每个数，且每个数只出现一次。以左上角大数独中第 1 行为例，可写成如下条件：

1111∨1121∨1131∨1141∨1151∨1161∨1171∨1181∨1191（第 1 行含有 1）

… …

1119∨1129∨1139∨1149∨1159∨1169∨1179∨1189∨1199（第 1 行含有 9）

-1111∨-1121（第 1 与第 2 格不同时为 1）

… …

-1111∨-1191（第 1 与第 9 格不同时为 1）

然后分别用 convert 函数对语义编码进行转换并加入 CNF 公式。此约束会产生 2×9×（9+9×$C\frac{2}{9}$）=5994 条子句。

③列约束：每列需要填入 1~9 中的每个数字，且每个数字只出现一次。这与行约束规则一致，转换规则也基本相同，共有 5994 条子句。

④3×3 的盒子约束：每个盒子要同时含有 1~9 中的每个数，且每个数字只出现一次。以左上角大数独左上角的盒子为例，可写成如下条件：

1111∨1121∨1131∨1211∨1221∨1231∨1311∨1321∨1331（包含 1）

… …

1119∨1129∨1139∨1219∨1229∨1239∨1319∨1329∨1339（包含 9）

-1111∨-1211（11 格与 21 格不同时为 1）

-1111∨-1311（11 格与 31 格不同时为 1）

… …

此约束会产生 2×9×（9+9×$C\frac{2}{9}$）=5994 条子句。

⑤中间的 9 个小格子共享两个数独，采用布尔等值公式可写出 9×18=162 个子句。

以上五种约束条件共会产生 24138 条子句，它们共同组成了双数独游戏通用的 CNF 约束条件子句集。

复杂度分析：时间复杂度、空间复杂度均为 O(1)。

（5）int solvesudoku(formula &newdouble)

函数功能：将 DPLL 求解的结果转移到棋盘数组内。

函数实现：遍历 DPLL 的结果数组，如果某变元取值为真，则将其改写为语义编码并写入棋盘数组内。

复杂度分析：由于变元数一定（1458 个），故时间复杂度、空间复杂度均为 O(1)。

（6）void printtwosudoku(formula &newdouble)

函数功能：打印双数独游戏棋盘。

函数实现：根据上面得到的棋盘数组，若某位置有解，则将其打印出来，否则打印字符“_”表示该处需要被填空。此函数需要注意行与列分界处的字符格式。

复杂度分析：只需遍历棋盘数组并循环打印分界线，时间复杂度、空间复杂度均为 O(1)。

（7）void digholes(formula &newdouble, int fullboard[300], int holecount)

函数功能：挖洞法生成双数独游戏。

函数实现：首先生成一个挖洞位置的随机数，判断挖去该数后填入其它 8 个数字是否仍能使双数独有解（加入这 8 个数代表的子句后，使用 DPLL 求解）。如果无解，说明挖去后仍有唯一解，可以挖去；否则重新生成随机数并检验。重复以上过程直到挖洞数量达到要求。

复杂度分析：设 h 为挖洞个数，则时间复杂度为 O(h)，空间复杂度为 O(1)。

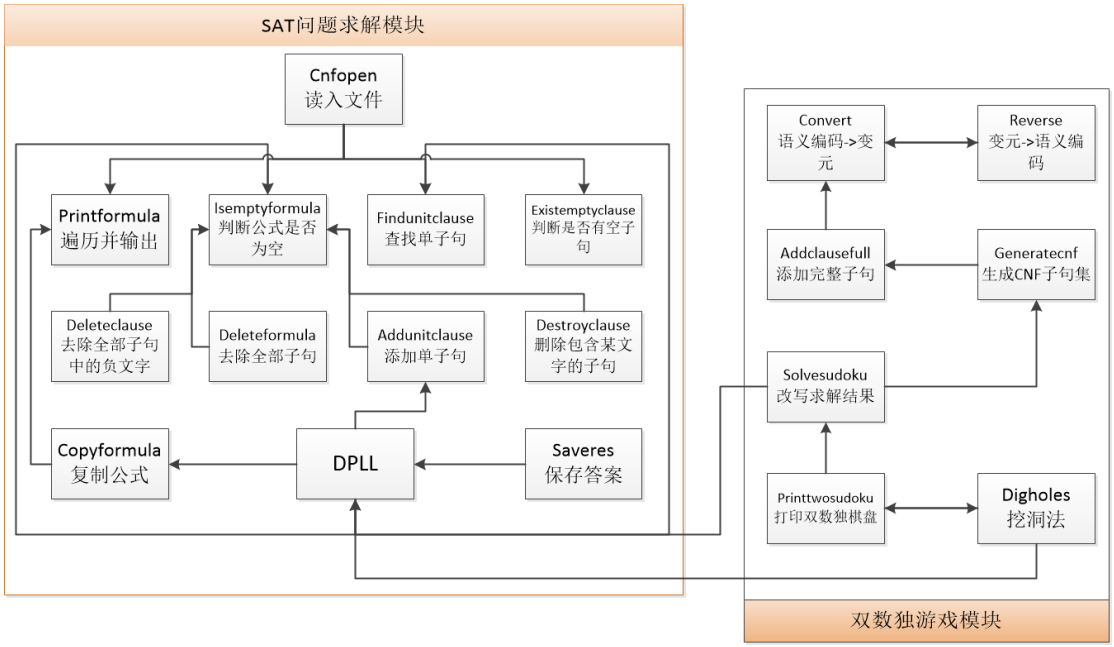

图 4-1 函数调用关系图

以上函数间的调用关系图如图 4-1 所示。全部程序代码详见附录。

4.2 系统测试

测试是判断程序是否达到预期效果的重要环节，有助于确保程序的质量并评估程序的效能。一般来说，测试可以分为白盒测试、黑盒测试、灰盒测试等多个

种类；而测试中的方法则体现出多样性，如等价类划分、边界值分析、判定表、因果图、正交试验、状态迁移、流程分析法（场景设计法）等[2]，需要根据测试的具体场景选择合适的方法。

对于本次课设的系统测试，主要采用灰盒测试的方式，具体过程如下：

4.2.1 SAT 问题求解模块系统测试

（一）模块的功能与设计目标

本模块目的在于能够正确读入并解析CNF算例文件，能对SAT问题进行求解并输出答案。

（二）测试数据的选取

为了探究模块能解出SAT问题规模的大小，验证DPLL算法的性能好坏，因此需要兼顾小、中、大算例。测试时从三种给定算例中各选择5~6个，进行DPLL算法求解并记录两种策略下的运行时间，并使用verify.exe程序对结果进行正确性检验。

（三）测试结果

（1）CNF文件读取与解析

首先读入“满足算例”下“S”文件夹中的算例“problem1-20.cnf”，系统提示“cnf文件读取完成，共包含20个文字，91个子句，0个单子句”，如图4-2所示。

接下来选择“遍历并输出子句”，测试解析验证功能。如图4-3所示，系统输出了公式内全部子句，可通过人工方式与CNF文件比对，二者显示的内容一致。

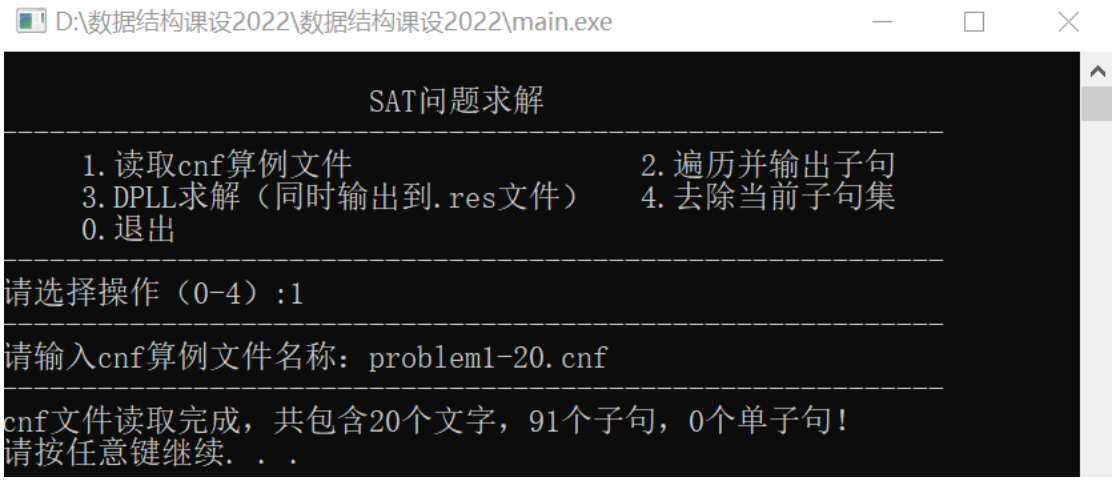

图 4-2 CNF 文件读入测试

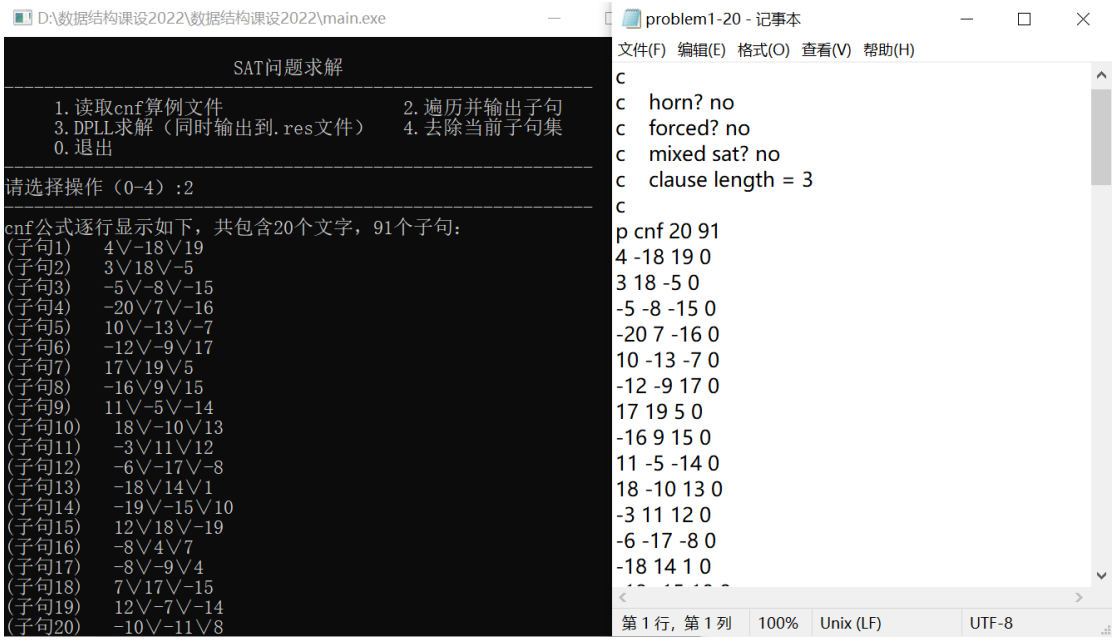

图4-3 CNF文件解析验证测试

（2）DPLL算法求解

选择“DPLL求解”，系统提示“请选择算法（1为原始算法，2为优化算法）”。分别输入“1”和“2”，均可得到一组正确解，同时系统输出求解过程所耗时间，如图4-4和图4-5所示。程序将.res文件输出到同一目录下后，使用verify.exe检验无误，如图4-6和图4-7所示。

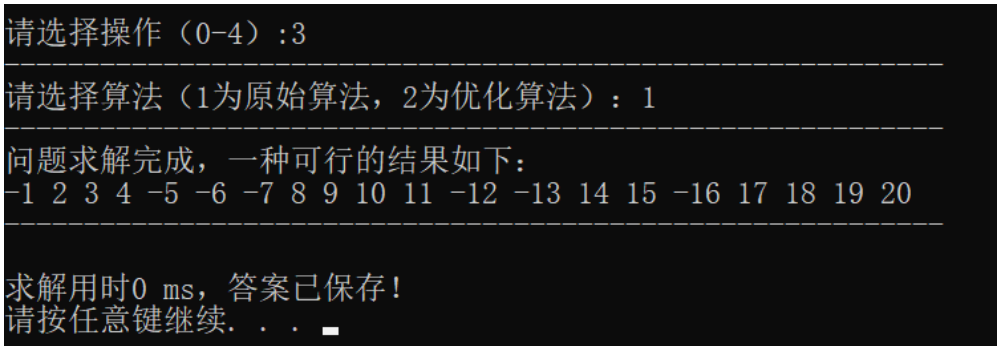

图4-4 DPLL原始算法求解可满足算例

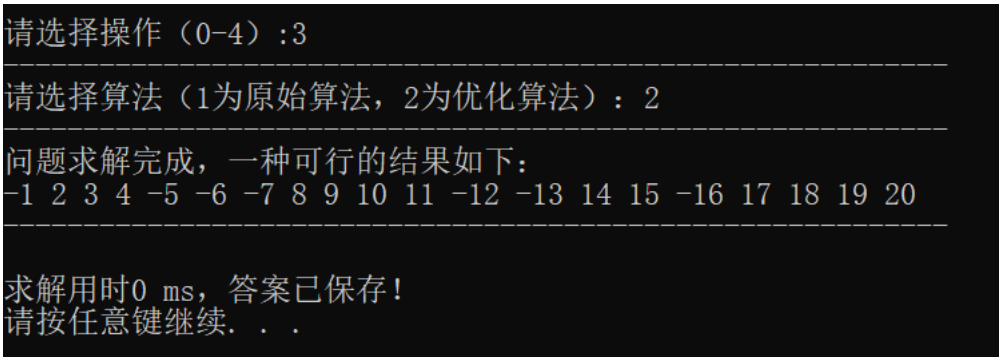

图4-5 DPLL优化算法求解可满足算例

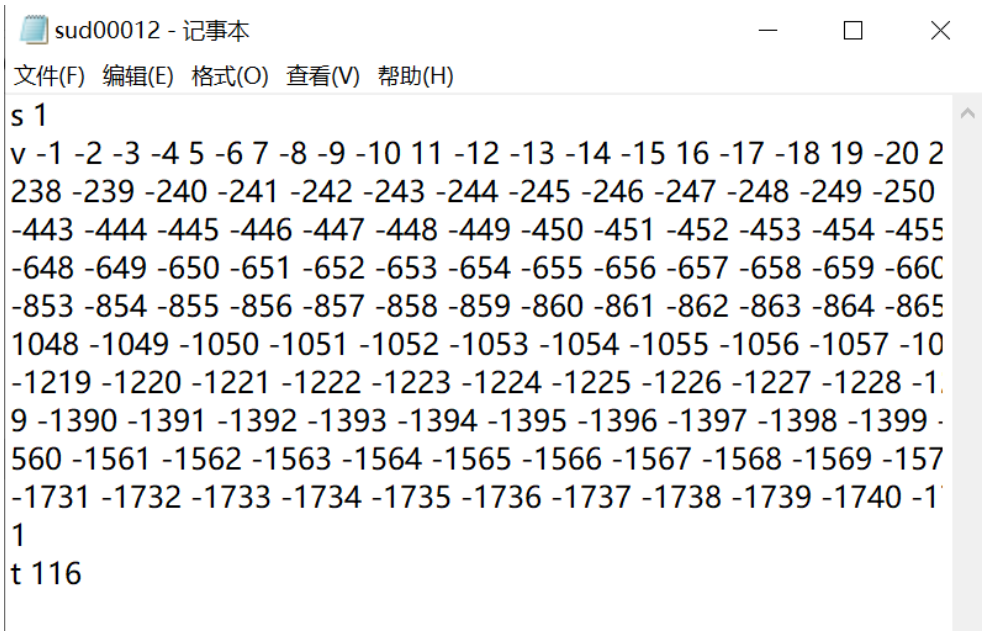

图4-6 存储结果的.res文件示例

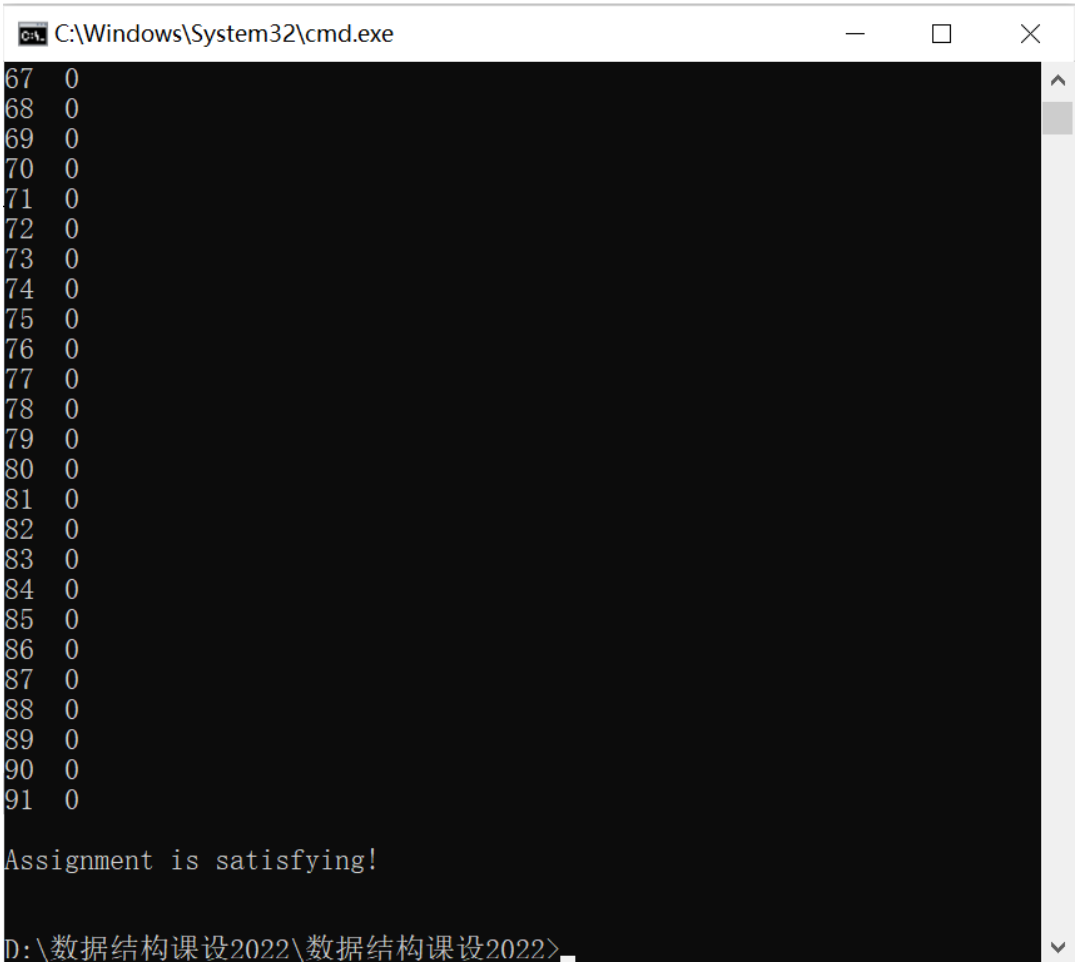

图4-7 使用verify.exe检验结果

对于不可满足算例，进行DPLL求解，系统输出“问题无解”，如图4-8所示。

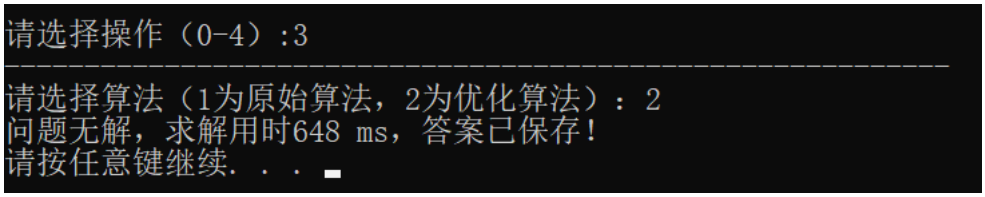

图4-8 DPLL优化算法求解不可满足算例

分别选取小（S）、中（M）、大（L）三种算例，求解结果如下：

算例名	规模	变元数	子句与变元数比值	求解结果	求解时间（ms）		优化率
					优化前	优化后	
problem1-20.cnf	S	20	4.55	可满足	0	0	-
problem2-50.cnf	S	50	1.6	可满足	30	0	100%
problem3-100.cnf	S	100	3.4	可满足	566	16	97.2%
problem6-50.cnf	S	50	2	可满足	84	0	100%
tst_v25_c100.cnf	S	25	4	可满足	0	0	-
problem11-100.cnf	S	100	6	可满足	132	0	100%
problem9-100.cnf	S	100	2	可满足	125	15	88%
problem5-200.cnf	M	200	1.6	可满足	60	23	61.7%
sud00001.cnf	M	301	9.24	可满足	54	36	33.3%
sud00009.cnf	M	303	9.41	可满足	84	0	100%
sud00012.cnf	M	232	8.19	可满足	116	47	59.5%
sud00079.cnf	M	301	9.34	可满足	37	16	56.8%
sud00082.cnf	M	224	7.87	可满足	83	7	91.6%
sud00861.cnf	M	297	9.16	可满足	47	30	36.2%
eh-dp04s04.shuffled-1075.cnf	L	1075	2.93	可满足	超时	1956	-
u-problem10-100.cnf	S	100	2	不可满足	39	9	76.9%
4（unsatisfied）.cnf	M	512	18.92	不可满足	15131	1409	90.7%
u-dp04u03.shuffled-825.cnf	L	825	2.92	不可满足	29621	648	97.8%

【注：优化率=（优化前时间-优化后时间）/优化前时间×100%】

（四）测试结果分析

根据以上结果，可以确定本模块达到了预期的需求，可以正常读入、解析CNF文件，运用DPLL算法对SAT问题进行求解并输出至.res文件中。值得一提的是，DPLL算法中改进后的策略在几乎所有测试算例中均发挥出色，对所有规模的算例都产生了极为明显的优化效果（在大规模算例中表现的尤为明显，例如有些算例使用原始算法需要上万毫秒才能得到结果，或甚至不能求出结果，但运用改进策略可以在1000ms的时间量级水平内得到正确结果）。

综上所述，本模块基本实现了一个高效率的初级SAT问题求解器。

4.2.2 双数独游戏模块系统测试

（一）模块的功能与设计目标

本模块目的在于实现 SAT 问题的应用，也就是将双数独游戏问题转化为 SAT 问题，实现双数独游戏的求解和生成，并具有一定的可玩性。

（二）测试数据的选取

为避免偶然性，双数独游戏的初始格局采用随机生成的棋盘（有空缺），其中包含一定量的初始提示数。

（三）测试结果

（1）双数独游戏求解

选择“求解双数独游戏”，系统展示一个初始双数独棋盘格局，如图4-9所示。

图4-9 双数独游戏求解-初始状态

然后随意按一个键，系统对该双数独CNF公式进行求解并输出结果，如图4-10所示。

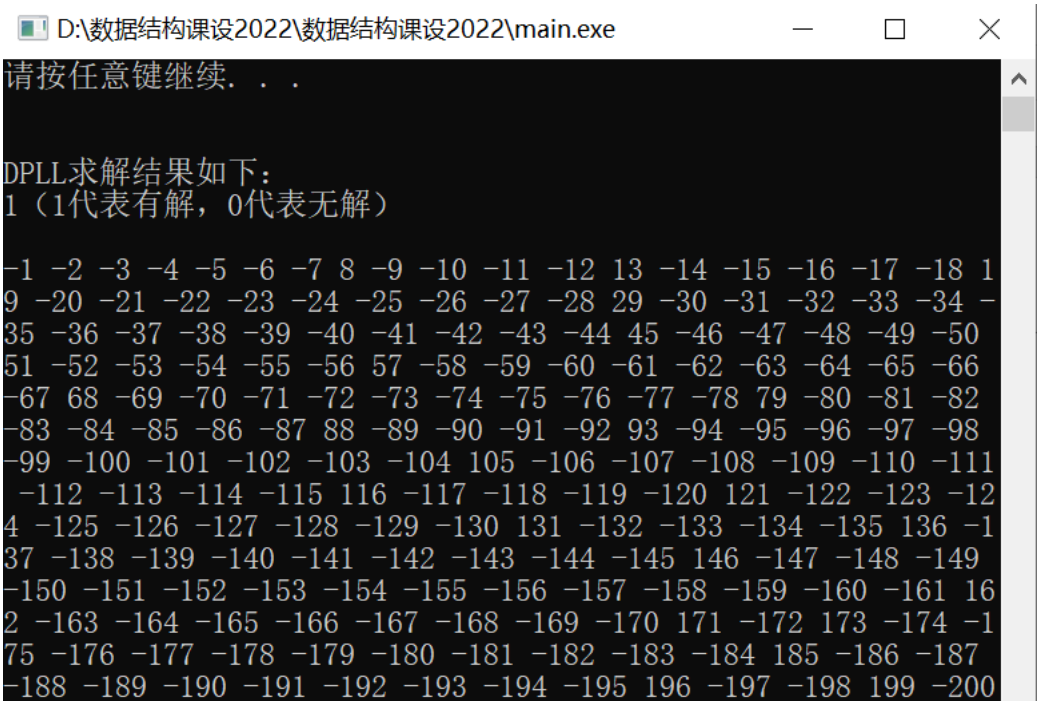

图4-10 DPLL算法求解双数独CNF公式

最后将结果转化为双数独的形式，并填入原双数独棋盘中，如图4-11所示。

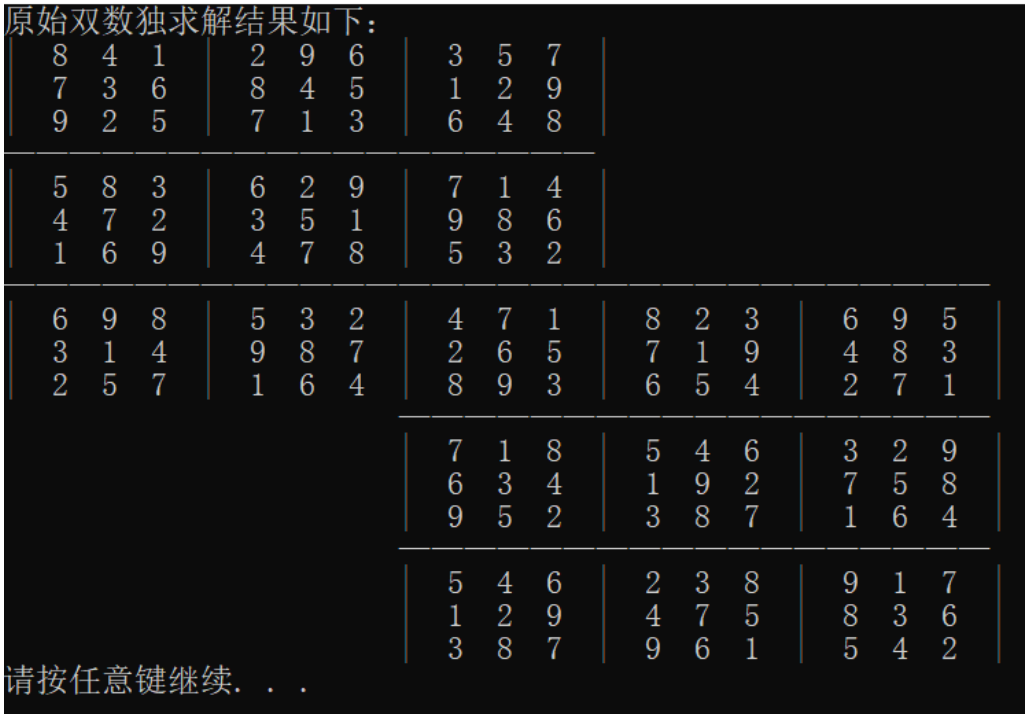

图4-11 双数独游戏求解-完成状态

（2）双数独游戏生成与游玩过程

选择“生成双数独”，系统提示“请输入挖洞数量”，输入“35”（此处不宜过多也不宜过少），系统输出生成好的新双数独棋盘，如图 4-12 所示。同时，系统提示“请输入填入数字的数独位置（左上角为 1，右下角为 2）、行、列和值，用空格隔开”，即游戏可以开始。

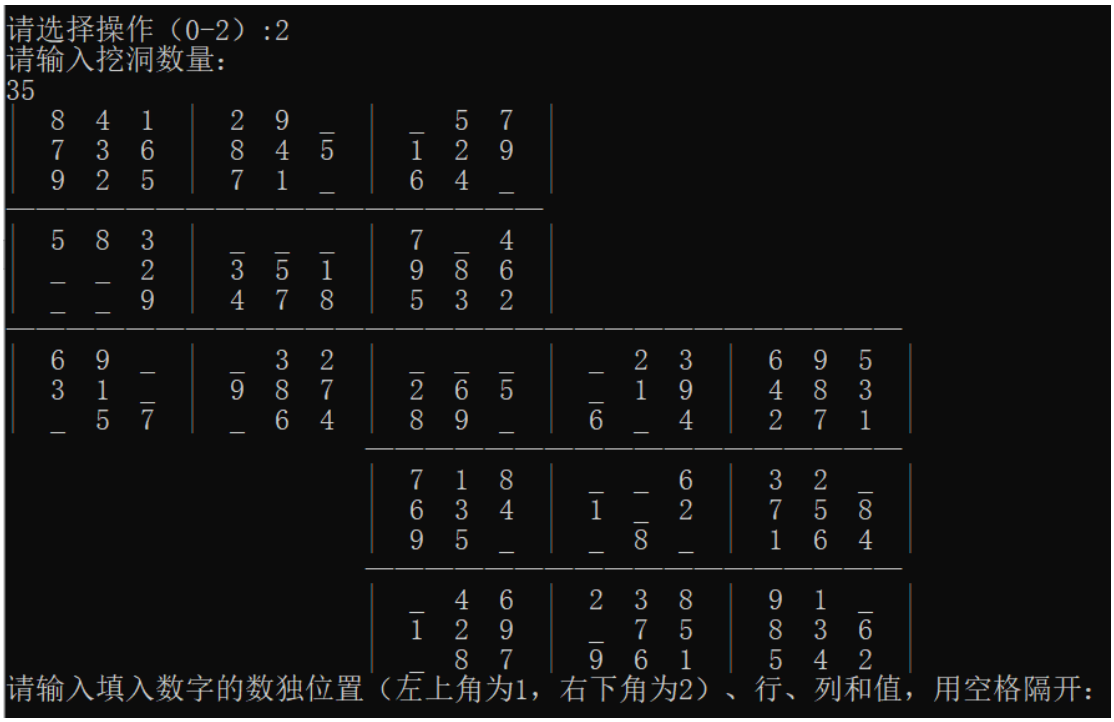

图 4-12 生成新双数独游戏

接下来用户可以自行操作。若对某一格输入正确（如图 4-12 中对左上角大数独第 1 行第 6 列位置输入 6，则输入“1 1 6 6”），系统提示“此方格输入正确”并将对应位置填入正确数字，如图 4-13、图 4-14 所示。否则，系统提示“输入有误，请重新输入”，如图 4-15 所示。

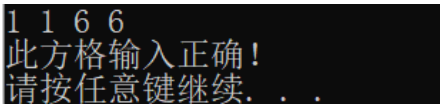

图 4-13 输入正确

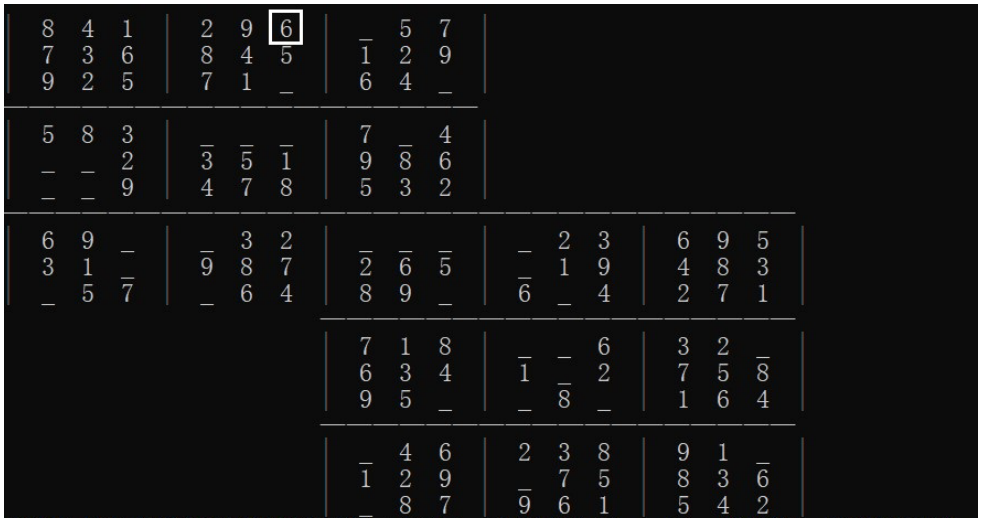

图 4-14 系统自动补全相应位置数字（见图中白框处）

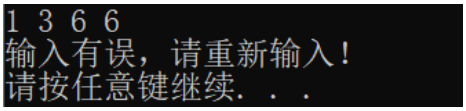

图 4-15 输入不正确

若用户正确填入了全部缺失数字，系统提示“数独求解正确”，游戏结束，如图 4-16 所示。

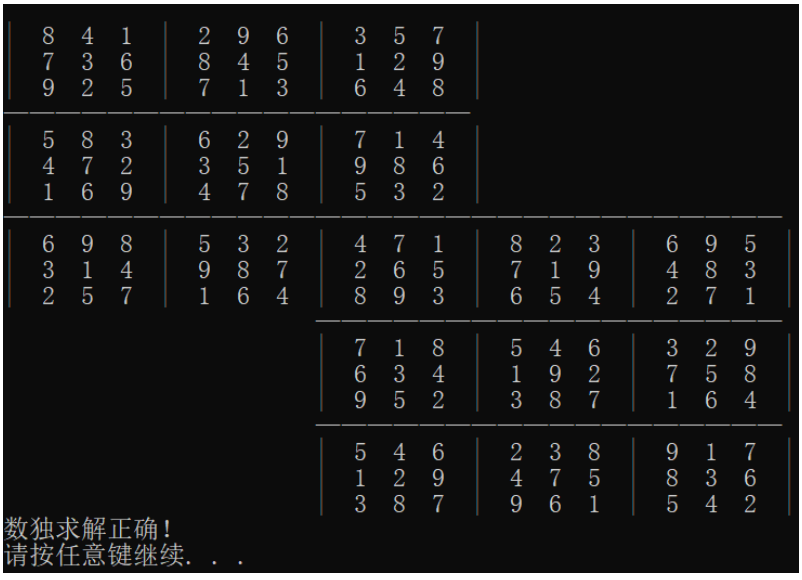

图 4-16 数独求解结束

（四）测试结果分析

根据测试可知双数独模块可以完成求解和生成双数独游戏两种任务。生成好新的双数独游戏后，可以根据输入的数值判断某位置填入是否正确，并实时给予反馈，棋盘也会随着游戏的进行而不断更新，具有充分的交互性。因此，本模块实现了SAT向应用问题的转换，将生活中的游戏进行理论化并达到了一定可玩性。

5 总结与展望

5.1 全文总结

在本次课程设计中，我的主要工作如下：

（1）认真阅读课设任务书，确定选题后查阅相关资料，了解了 SAT 问题的概念、研究方法、在国际上的地位与研究现状，以及 DPLL 算法的思想内核；认识到双数独游戏和传统数独游戏的区别与联系，还有将该应用问题归结为 SAT 问题的方法。

（2）根据 CNF 公式的数据特征，利用 C 语言定义了合适的数据结构；根据系统需求，设计了系统总体结构与流程图。

（3）单独用一个文件来存放基本的数据结构等相关定义，而对程序的两个主要模块（SAT 问题求解和双数独游戏模块）中需要调用的函数分别进行了编写，实现了函数的模块化表示。

（4）查找了 DPLL 算法的常见优化策略及其效果，并结合自行数学推理和分析，逐渐打磨出了程序中所使用的的改进策略。

（5）实现了基于 SAT 问题求解器的双数独游戏程序，其具有一定的可玩性与交互性。

（6）给程序代码添加了足量的注释，确保其可读性强、足够规范。

（7）进行了大量的调试与测试工作，改正了很多原有的漏洞。

（8）按照规范撰写了课程设计报告，对课设的全过程作了回顾与思考。

综上所述，在不断探索、试错与改进的过程中，我最终圆满完成了课设任务，实现了基于 SAT 的双数独游戏求解程序，并通过了课堂验收。

5.2 未来展望

本次课设给我带来了宝贵的经验。在今后的学习（尤其是计算机专业核心课程）中，我将围绕着如下两个方面开展。

（1）认识到理论是应用的基础，应用是理论的价值，任何理论课程都需要在实践中提高自己的动手能力，从而积累求解综合性应用问题的经验。

（2）经过此次课设，我基本了解了程序开发的通用工程流程，并形成了初步的工程化素养。因此我可以尝试在后续的大创或科研项目中应用规范化的分析能力，不断提升自己的综合能力。

6 体会

本次课设在 2022 年 9 月上旬，也就是开学的两个星期之内完成。用一句话概括这个过程就是：虽然比较累，但收获良多。

在上学期的暑假里我已经对问题的背景有了一些了解，但是没有写太多的代码，只是做了一些准备工作。直到开学后真正上手核心模块的代码后才发现有很多细节问题需要注意。例如我在写“复制 CNF 公式”、“删除子句”等函数时，调试中遇到过很多错误返回值如“3221225477”“3221225725”等，主要原因在于可能一时疏忽没有时刻把指针域置为 NULL 导致出现野指针，或者是循环中没有特判一些特殊情况导致循环无法跳出导致内存越界。这种问题往往十分隐蔽，所以查错的过程比较煎熬，但却能给我深刻的教训。还有在双数独游戏里，生成约束子句的函数需要多重循环，这里也容易因为一些小错（循环范围写错、i 和 j 搞反、没考虑重叠部分共用等）使得生成的 CNF 子句集不满足要求，从而导致 DPLL 算法求解失败。

可以说，本次课设是对上学期的数据结构与数据结构实验的一种承接与拓展。首先，问题中涉及到的 CNF 公式、子句、文字实质上是链表与链表结点的应用。其次，DPLL 算法本质上是基于二叉搜索树的回溯算法。而改进后的策略需要对全部文字的出现次数进行排序，这也是数据结构课程中学到的知识（有很多排序算法）。因此，这是一次对课程内容的补充和升华，也让我真正体验到了完整的工程化程序设计过程。

最后，我要感谢在整个过程中对我提供过帮助的同学和老师，也要感谢不断在失败中成长的自己！

参考文献

[1] SAT 问题简介 https://zhuanlan.zhihu.com/p/432853785

[2] 软件测试常用的七大方法

https://blog.csdn.net/qq_38913292/article/details/119315366

附录

附录一 程序源代码

（一）main.cpp

```
#include "definition.h"
#include "SATsolver.cpp"
#include "twosudoku.cpp"

int SATsolver() {
    int op = 1, strategy; //记录用户输入
    char input[50];
    char output[50];//输入输出文件名
    formula original;
    original.clausenum = 0;
    original.literalnum = 0;
    original.singleclausenum = 0;
    original.currentclausenum = 0;
    original.head = NULL;//初始化 CNF 公式
    while (op) {
        system("cls");
        printf("\n                                                    SAT  问 题 求 解
\n");
        printf("------------------------------------------------------------\n");
        printf("        1.读取 cnf 算例文件                          2.遍历并输出子句
\n");
        printf("        3.DPLL 求解（同时输出到.res 文件）      4.去除当前子句集
\n");
        printf("        0.退出\n");
        printf("------------------------------------------------------------\n");
        printf("请选择操作（0-4）:");
        scanf("%d", &op);
        switch (op) {
```

```
case 1:
    if (original.head != NULL) {
        printf("------------------------------------------------------------\n");
        printf("已存在 cnf 算例文件！\n");
        system("pause");
        break;
    } else {
        printf("------------------------------------------------------------\n");
        printf("请输入 cnf 算例文件名称：");
        scanf("%s", input);
        strcpy(output, input);
        output[strlen(output) - 3] = 'r';
        output[strlen(output) - 2] = 'e';
        output[strlen(output) - 1] = 's';
        cnfopen(original, input);
        system("pause");
        break;
    }
case 2:
    if (original.head == NULL) {
        printf("------------------------------------------------------------\n");
        printf("不存在 cnf 算例文件！\n");
        system("pause");
        break;
    } else {
        printformula(original);
        system("pause");
        break;
    }
case 3:
    if (original.head == NULL) {
        printf("------------------------------------------------------------\n");
        printf("不存在 cnf 算例文件！\n");
```

```
                    system("pause");
                    break;
                } else {
                    printf("-----------------------------------------------------------\n");
                    printf("请选择算法（1 为原始算法，2 为优化算法）：");
                    scanf("%d", &strategy);
                    clock_t ta = clock();//记录算法开始时间
                    if (DPLL(original, strategy) == OK) {
                        clock_t tb = clock();//记录算法结束时间

    printf("-----------------------------------------------------------\n");
                        printf("问题求解完成，一种可行的结果如下：\n");
                        for (int i = 1; i <= original.literalnum; i++) {
                            if (result[i])
                                printf("%d ", i);
                            else
                                printf("-%d ", i);
                        }
                        saveres(output, original, (tb - ta) * 1000 /
CLOCKS_PER_SEC, 1);

    printf("\n-----------------------------------------------------------\n");
                        printf("\n 求解用时%d ms，答案已保存！\n", (tb - ta) *
1000 / CLOCKS_PER_SEC);
                    } else {
                        clock_t tc = clock();
                        saveres(output, original, (tc - ta) * 1000 /
CLOCKS_PER_SEC, 0);
                        printf("问题无解，求解用时%d ms，答案已保存！\n", (tc
- ta) * 1000 / CLOCKS_PER_SEC);
                    }
                    system("pause");
                    break;
```

```
                }
            case 4:
                if (original.head == NULL) {
                    printf("------------------------------------------------------------\n");
                    printf("不存在 cnf 算例文件！\n");
                    system("pause");
                    break;
                } else {
                    deleteformula(original);
                    system("pause");
                    break;
                }
            case 0:
                return 0;
        }
    }
    return 0;
}

int twosudoku() {
    int op = 1, ans = 0, holecount = 0; //记录用户输入、DPLL 求解结果和挖洞数量
    int a, x, y, value; //记录第 a 个数独 x 行 y 列填入的值 value
    formula newdouble;
    newdouble.clausenum = 0;
    newdouble.literalnum = 0;
    newdouble.singleclausenum = 0;
    newdouble.currentclausenum = 0;
    newdouble.head = NULL;//初始化双数独 CNF
    while (op) {
        system("cls");
        printf("\n                                                    双 数 独 游 戏
\n");
        printf("------------------------------------------------------------\n");
```

```
printf("                1.求解双数独游戏                    2.生成双数独\n");
printf("                0.退出\n");
printf("-------------------------------------------------------\n");
printf("请选择操作（0-2）：");
scanf("%d", &op);
switch (op) {
    case 1:
        generatecnf(newdouble);
        printf("下面展示初始双数独棋盘格局：\n");
        printtwosudoku1(newdouble);
        system("pause");
        printf("\n\nDPLL 求解结果如下：\n");
        ans = solvesudoku(newdouble);
        printf("%d（1 代表有解，0 代表无解）\n\n", ans);
        if (ans)
            for (int i = 1; i <= newdouble.literalnum; i++) {
                if (result[i])
                    printf("%d ", i);
                else
                    printf("-%d ", i);
            }
        system("pause");
        printf("\n\n 原始双数独求解结果如下：\n");
        printtwosudoku2(newdouble);
        system("pause");
        break;
    case 2:
        if (!fullboard[111]) {
            printf("不存在初始棋盘，请先求解！\n");
            system("pause");
            break;
        }
```

```
                printf("请输入挖洞数量：\n");
                scanf("%d", &holecount);
                digholes(newdouble, fullboard, holecount);//挖洞
                printtwosudoku2(newdouble);
                while (holecount > 0) {//游玩过程
                    printf("请输入填入数字的数独位置（左上角为 1，右下角为
2）、行、列和值，用空格隔开：\n");
                    scanf("%d %d %d %d", &a, &x, &y, &value);
                    if (!board[a * 100 + x * 10 + y] && fullboard[a * 100 + x * 10
+ y] == value) {
                        board[a * 100 + x * 10 + y] = value;
                        holecount--;
                        printf("此方格输入正确！\n");
                    } else
                        printf("输入有误，请重新输入！\n");
                    system("pause");
                    system("cls");
                    printf("\n\n\n\n\n\n");
                    printtwosudoku2(newdouble);
                }
                printf("数独求解正确！\n");
                system("pause");
                break;
            case 0:
                return 0;
        }
    }
    return 0;
}

int main() {
    int op = 1;
    while (op) {
```

```
        system("cls");
        printf("\n                    基于 SAT 的双数独游戏求解程序
\n");
        printf("\n                大数据 2101 班   李嘉鹏   U202115652
\n");
        printf("------------------------------------------------------------\n");
        printf("          1.SAT 问题求解          2.生成双数独\n");
        printf("          0.退出\n");
        printf("------------------------------------------------------------\n");
        printf("请选择操作（0-2）：");
        scanf("%d", &op);
        switch (op) {
            case 1:
                SATsolver();
                system("pause");
                break;
            case 2:
                twosudoku();
                system("pause");
                break;
            case 0:
                return 0;
        }
    }
    return 0;
}
```

（二）definition.h

```
#ifndef data_structure_project_2022
#define data_structure_project_2022
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <math.h>
#include <time.h>
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

int result[3500] = {0}; //各文字取值的结果数组，1 代表真，0 代表假
int found[3500] = {0}; //单子句的查找记录数组
int temp[50] = {0}; //创建双数独 cnf 时暂存读入的数
int board[300] = {0}; //存储棋盘上数字的解，board[abc]表示第 a 个大数独中 b 行 c 列的值
int fullboard[300] = {0}; //存储初始时的合法棋盘解（用于挖洞），全过程保持不变
FILE *fp;

typedef struct literal { //以链表形式存储文字
    int word;//正值代表正文字，反之为负文字
    struct literal *next;//下一个文字
} literal, *L;

typedef struct clause { //以链表形式存储子句
    int num; //某子句内的文字总数
    struct literal *head; //第一个文字
    struct clause *next; //下一个子句
} clause, *C;

typedef struct formula { //整个公式
    int clausenum;//文字总数
    int literalnum;//子句总数
    int emptyclausenum;//空子句总数
```

```
    int singleclausenum;//单子句总数（随 DPLL 进行而减少）
    int currentclausenum;//当前子句总数（随 DPLL 进行而减少）
    struct clause *head;//第一个子句
} formula, *F;
#endif
```

（三）SATsolver.cpp

```
#ifndef data_structure_project_2022_1
#define data_structure_project_2022_1
#include "definition.h"

void cnfopen(formula &original, char *input) {//读入 CNF 文件
    fp = fopen(input, "r");
    if (!fp)
        printf("cnf 文件读取失败！\n");
    else {
        char c;
        int literal_read, literalnum_in_the_clause;
        int literal_read_temp[3500];//存储读入的文字
        literal *frontl = NULL;
        clause *frontc = NULL;
        while (fread(&c, sizeof(char), 1, fp))//不断读取字符，直到遇到 p 为止
            if (c == 'p')
                break;
        fread(&c, sizeof(char), 5, fp);
        fscanf(fp, "%d", &original.literalnum);//读入文字数
        fscanf(fp, "%d", &original.clausenum);//读入子句数
        original.currentclausenum = original.clausenum;
        original.emptyclausenum = 0;
        for (int i = 1; i <= original.clausenum; i++) {
            literalnum_in_the_clause = 0;
            clause *newclause = (clause *)malloc(sizeof(clause));//创建新子句
```

```
            if (i > 1)
                frontc->next = newclause;
            frontc = newclause;
            if (i == 1)
                original.head = newclause;//确定公式的头子句
            while (fscanf(fp, "%d", &literal_read) != EOF && literal_read)
                literal_read_temp[++literalnum_in_the_clause] = literal_read;
            if (literalnum_in_the_clause > 0) {
                if (literalnum_in_the_clause == 1)
                    original.singleclausenum++;
                literal *head = (literal *)malloc(sizeof(literal));
                head->word = literal_read_temp[1];
                head->next = NULL;
                newclause->head = head;
                frontl = head;
                for (int j = 2; j <= literalnum_in_the_clause; j++) { //生成一条子句
链表
                    literal *newliteral = (literal *)malloc(sizeof(literal));
                    newliteral->word = literal_read_temp[j];
                    newliteral->next = NULL;
                    frontl->next = newliteral;
                    frontl = newliteral;
                }
            }
            newclause->next = NULL;
            newclause->num = literalnum_in_the_clause;
            literalnum_in_the_clause = 0;
        }
        printf("------------------------------------------------------------\n");
        printf("cnf 文件读取完成，共包含%d 个文字，%d 个子句，%d 个单子句！
\n", original.literalnum, original.clausenum,
                original.singleclausenum);
    }
```

```
    fclose(fp);
}

void printformula(formula &original) {//遍历并输出公式
    clause *tempc = original.head;
    literal *templ = NULL;
    int clausecount = 0;
    printf("------------------------------------------------------------\n");
    printf("cnf 公式逐行显示如下，共包含%d 个文字，%d 个子句：\n",
original.literalnum, original.clausenum);
    while (tempc->next != NULL) {
        templ = tempc->head;
        printf("(子句%d)      ", ++clausecount);
        while (templ->next != NULL) {
            printf("%d∨", templ->word);
            templ = templ->next;
        }
        printf("%d\n", templ->word);
        tempc = tempc->next;
    }
    templ = tempc->head;
    printf("(子句%d)      ", ++clausecount);
    while (templ->next != NULL) {
        printf("%d∨", templ->word);
        templ = templ->next;
    }
    printf("%d\n", templ->word);
    printf("------------------------------------------------------------\n");
    printf("遍历完成！\n");
}

void deleteformula(formula &original) {//删除整个公式
    clause *deletec = original.head, *deletetempc;
```

```
    while (deletec != NULL) {//从头子句向后搜索
        literal *deletel = deletec->head, *deletetempl;
        while (deletel != NULL) {//从头文字向后搜索
            deletetempl = deletel->next;
            free(deletel);
            deletel = deletetempl;
        }
        deletetempc = deletec->next;
        free(deletec);
        deletec = deletetempc;
    }
    original.head = NULL;
    original.clausenum = 0;
    original.literalnum = 0;
    original.singleclausenum = 0;
    original.currentclausenum = 0;//初始化
}

int findunitclause(formula &original) {//查找单子句
    clause *find = original.head;
    if (!original.singleclausenum)//公式中没有单子句
        return 0;
    while (find != NULL) {
        if (find->num == 1)
            return find->head->word;
        find = find->next;
    }
    return 0;
}

void destroyclause(formula &original, int unitclause) { //删除所有包含某文字的子句
    clause *findc = original.head, *front;
```

```
literal *findl = findc->head;
while (findl != NULL && findc == original.head) {
    if (findl->word == unitclause) {
        original.head = findc->next; //删除第一个子句
        if (findc->num == 1)
            original.singleclausenum--;//单子句数量-1
        findc = findc->next;
        if (findc != NULL)
            findl = findc->head;
        else
            findl = NULL;
        original.currentclausenum--;
    } else
        findl = findl->next;
}
front = findc;
if (findc != NULL)
    findc = findc->next;
while (findc != NULL) { //遍历余下所有子句
    findl = findc->head;
    while (findl != NULL) {
        if (findl->word == unitclause) {
            front->next = findc->next;
            original.currentclausenum--;
            if (findc->num == 1)
                original.singleclausenum--;//单子句数量-1
            break;
        }
        findl = findl->next;
    }
    if (findl == NULL)
        front = front->next;
    findc = findc->next;
```

```
    }
}

void deleteclause(formula &original, int negative) { //去除子句中的负文字
    int flag;//若在某子句内找到负文字则为 1，否则为 0
    clause *findc = original.head;
    literal *findl = original.head->head, *front;
    while (findc != NULL) {
        flag = 0;
        findl = findc->head;
        while (findl != NULL) {
            if (findl->word == negative) {
                flag = 1;
                findc->num--;
                if (!findc->num)
                    original.emptyclausenum++;
                if (findl != findc->head) {
                    front = findc->head;
                    while (front->next != findl)
                        front = front->next;
                    front->next = findl->next;
                    free(findl);
                    front = front->next;
                    findl = front;
                } else {
                    if (findl->next == NULL) {
                        free(findc->head);
                        findc->head = NULL;
                        break;
                    } else {
                        findc->head = findl->next;
                        free(findl);
                        findl = findc->head;
```

```
                    }
                }
                continue;
            }
            findl = findl->next;
        }
        if (findc->num == 1 && flag == 1)
            original.singleclausenum++;
        findc = findc->next;
    }
}

int existemptyclause(formula &original) {//判断是否存在空子句
    clause *findc = original.head;
    while (findc != NULL) {
        if (!findc->num)
            return TRUE;
        findc = findc->next;
    }
    return FALSE;
}

int isemptyformula(formula &original) {//判断公式是否为空
    if (!original.currentclausenum)
        return TRUE;
    return FALSE;
}

void copyformula(formula &temp, formula &original) {//复制整个公式，边读边写
    temp.clausenum = original.clausenum;
    temp.currentclausenum = original.currentclausenum;
    temp.literalnum = original.literalnum;
    temp.singleclausenum = original.singleclausenum;
```

```
    temp.emptyclausenum = original.emptyclausenum;//初始化，复制数据项
    clause *copyc = original.head;
    literal *copyl;//两个 copy 指针指在原链表上
    literal *frontl = NULL;
    clause *frontc = NULL;//两个 front 指针指在新链表上
    while (copyc != NULL) {
        clause *newclause = (clause *)malloc(sizeof(clause));
        newclause->num = copyc->num;
        copyl = copyc->head;
        while (copyl != NULL) {
            literal *newliteral = (literal *)malloc(sizeof(literal));
            newliteral->word = copyl->word;
            if (copyl == copyc->head) {
                newclause->head = newliteral;
                frontl = newliteral;
            } else {
                frontl->next = newliteral;
                frontl = frontl->next;
            }
            newliteral->next = NULL;
            copyl = copyl->next;
        }
        if (copyc == original.head) {
            frontc = newclause;
            temp.head = newclause;
        } else {
            frontc->next = newclause;
            frontc = frontc->next;
        }
        newclause->next = NULL;
        copyc = copyc->next;
    }
}
```

```
void addunitclause(formula &original, int key) { //在原公式中添加单子句
    clause *newclause = (clause *)malloc(sizeof(clause));
    if (original.head == NULL) {
        original.head = newclause;
        newclause->next = NULL;
    } else {
        newclause->next = original.head;
        original.head = newclause;
    }
    literal *singleliteral = (literal *)malloc(sizeof(literal));
    singleliteral->word = key;
    singleliteral->next = NULL;
    newclause->head = singleliteral;
    newclause->num = 1;
    original.currentclausenum++;//当前子句数+1
    original.singleclausenum++;//单子句数+1
}

int DPLL(formula &original, int strategy) {//DPLL 算法，两种策略合一
    int unitclause = findunitclause(original), key, max = 0;
    int countposi[3500] = {0}, countnega[3500] = {0}; //分别统计正负文字出现次
数
    while (unitclause) {//不断找单子句
        if (unitclause > 0)
            result[unitclause] = 1;//正文字取真
        else
            result[-unitclause] = 0; //负文字取假
        destroyclause(original, unitclause); //从公式中删除所有包含该文字的子句
        if (isemptyformula(original))
            return OK;//判断现在是否为空集
        deleteclause(original, -unitclause); //删除所有子句中的负文字
```

```
    if (existemptyclause(original)) //判断现在是否有空子句（不可满足）
        return ERROR;
    unitclause = findunitclause(original);
}
formula temp;
copyformula(temp, original); //创建新公式 temp 用来在 DPLL 过程中分支
if (strategy == 1) { //改进前策略，按照从前到后的顺序选取变元 key
    key = original.head->head->word;
    addunitclause(original, key);//在原公式中加入新变元
    addunitclause(temp, -key);//在原公式中加入负的新变元
} else if (strategy == 2) { //改进后策略，选取出现次数最多的 key
    clause *searchc = original.head;
    literal *searchl;
    while (searchc != NULL) {
        searchl = searchc->head;
        while (searchl != NULL) {
            if (searchl->word > 0)
                countposi[searchl->word]++;
            else
                countnega[-searchl->word]++;
            searchl = searchl->next;
        }
        searchc = searchc->next;
    }
    for (int i = 1; i <= original.literalnum; i++)
        if (max < countposi[i]) {
            key = i;
            max = countposi[i];
        }
    if (!max)//不存在正文字，所以找出现次数最多的负文字
        for (int i = 1; i <= original.literalnum; i++)
            if (max < countnega[i]) {
                key = i;
```

```
                max = countnega[i];
            }
        addunitclause(original, key);//在原公式中加入新变元
        addunitclause(temp, -key);//在原公式中加入负的新变元
    }
    if (DPLL(original, strategy))//分裂策略
        return OK;
    else
        return DPLL(temp, strategy);
}

void saveres(char *output, formula &original, int time, int DPLLstatus) {//将结果输出到.res 文件
    if ((fp = fopen(output, "w")) == NULL)
        printf("保存失败！\n");
    else {
        fprintf(fp, "s %d\nv ", DPLLstatus);
        for (int i = 1; i <= original.clausenum; i++) {
            if (result[i] > 0)
                fprintf(fp, "%d ", i);
            else
                fprintf(fp, "-%d ", i);
        }
        fprintf(fp, "\n");
        fprintf(fp, "t %d", time);
    }
    fclose(fp);
}
#endif
```

（四）twosudoku.cpp

```
#include "definition.h"
#include "SATsolver.cpp"
```

/*列举双数独游戏中的变元编码：

格式：abcd a 左上/右下大格子 b/c 行/列位置 d 填入的数

数独 1：

1111 1112 1113 ... 1119

1121 1122 1123 ... 1129

...

1191 1192 1193 ... 1199

...

1991 1992 1993 ... 1999 共 9*9*9=729 个 literal

数独 2：同上，第一位改成 2，共 729 个 literal。因此总共有 1458 个变元

对于重叠部分：每个格对应 18 条子句

对于 literal 的表示，设计转换公式：abcd=(a-1)*729+(b-1)*81+(c-1)*9+d

以下循环中，四位数格式为 aijd

*/

clause *frontc = NULL; //记录复制过程中子句的位置

```
int convert(int x) {//将四位数语义编码从 1 到 1458 连续编码
    int a = x / 1000, i = (x - a * 1000) / 100, j = (x - a * 1000 - i * 100) / 10, d = x %
10;
    return (a - 1) * 729 + (i - 1) * 81 + (j - 1) * 9 + d;
}

int reverse(int x) { //将变元转化为 abcd 的形式
    int a, b, c, d;
    a = (x - 1) / 729 + 1;
    b = (x - (a - 1) * 729 - 1) / 81 + 1;
    c = (x - (a - 1) * 729 - (b - 1) * 81 - 1) / 9 + 1;
    d = x - (a - 1) * 729 - (b - 1) * 81 - (c - 1) * 9;
    return a * 1000 + b * 100 + c * 10 + d;
}
```

```
void addclausefull(formula &newdouble, int count, int temp[50]) {
    //添加完整的子句，count 代表句中变元数量
    clause *newclause = (clause *)malloc(sizeof(clause));
    literal *frontl = NULL;
    newclause->next = NULL;
    newclause->num = count;
    for (int i = 1; i <= count; i++) {
        literal *newliteral = (literal *)malloc(sizeof(literal));
        newliteral->word = temp[i];
        newliteral->next = NULL;
        if (i == 1) {
            newclause->head = newliteral;
            frontl = newliteral;
        } else {
            frontl->next = newliteral;
            frontl = newliteral;
        }
    }
    if (newdouble.head == NULL) {
        newdouble.head = newclause;
        frontc = newclause;
    } else {
        frontc->next = newclause;
        frontc = frontc->next;
    }
    newdouble.clausenum++;
    newdouble.currentclausenum++;
    if (count == 1)
        newdouble.singleclausenum++;
}

void generatecnf(formula &newdouble) {//生成双数独约束条件 CNF 子句集
```

```
newdouble.clausenum = 0;
newdouble.currentclausenum = 0;
newdouble.emptyclausenum = 0;
newdouble.literalnum = 1458;
newdouble.singleclausenum = 0;
newdouble.head = NULL; //初始化
//加入初始提示数信息
addunitclause(newdouble, convert(1173));
addunitclause(newdouble, convert(1185));
addunitclause(newdouble, convert(1217));
addunitclause(newdouble, convert(1236));
addunitclause(newdouble, convert(1254));
addunitclause(newdouble, convert(1299));
addunitclause(newdouble, convert(1322));
addunitclause(newdouble, convert(1384));
addunitclause(newdouble, convert(1415));
addunitclause(newdouble, convert(1446));
addunitclause(newdouble, convert(1469));
addunitclause(newdouble, convert(1514));
addunitclause(newdouble, convert(1588));
addunitclause(newdouble, convert(1626));
addunitclause(newdouble, convert(1639));
addunitclause(newdouble, convert(1668));
addunitclause(newdouble, convert(1675));
addunitclause(newdouble, convert(1716));
addunitclause(newdouble, convert(1729));
addunitclause(newdouble, convert(1753));
addunitclause(newdouble, convert(1813));
addunitclause(newdouble, convert(1834));
addunitclause(newdouble, convert(2148));
addunitclause(newdouble, convert(2195));
addunitclause(newdouble, convert(2247));
addunitclause(newdouble, convert(2274));
```

```
addunitclause(newdouble, convert(2387));
addunitclause(newdouble, convert(2445));
addunitclause(newdouble, convert(2482));
addunitclause(newdouble, convert(2499));
addunitclause(newdouble, convert(2523));
addunitclause(newdouble, convert(2643));
addunitclause(newdouble, convert(2658));
addunitclause(newdouble, convert(2667));
addunitclause(newdouble, convert(2724));
addunitclause(newdouble, convert(2781));
addunitclause(newdouble, convert(2797));
addunitclause(newdouble, convert(2839));
addunitclause(newdouble, convert(2896));
addunitclause(newdouble, convert(2913));
addunitclause(newdouble, convert(2928));
addunitclause(newdouble, convert(2949));
frontc = newdouble.head;
while (frontc->next != NULL)
    frontc = frontc->next;

//每个单元格只能填入唯一一个数字,此处将中间重叠的 9 个格视为两个单独的格子
for (int a = 1; a <= 2; a++)
    for (int i = 1; i <= 9; i++)
        for (int j = 1; j <= 9; j++) {
            for (int d = 1; d <= 9; d++) {
                temp[d] = (a - 1) * 729 + (i - 1) * 81 + (j - 1) * 9 + d;
            }
            addclausefull(newdouble, 9, temp);
            for (int m = 1; m <= 8; m++)
                for (int n = m + 1; n <= 9; n++) {
                    temp[1] = -((a - 1) * 729 + (i - 1) * 81 + (j - 1) * 9 + m);
                    temp[2] = -((a - 1) * 729 + (i - 1) * 81 + (j - 1) * 9 + n);
```

```
                    addclausefull(newdouble, 2, temp);
                }
            }

//行约束
for (int a = 1; a <= 2; a++)
    for (int i = 1; i <= 9; i++) {
        for (int d = 1; d <= 9; d++) {
            for (int j = 1; j <= 9; j++)
                temp[j] = (a - 1) * 729 + (i - 1) * 81 + (j - 1) * 9 + d;
            addclausefull(newdouble, 9, temp);
        }
        for (int d = 1; d <= 9; d++)
            for (int j = 1; j <= 8; j++)
                for (int k = j + 1; k <= 9; k++) {
                    temp[1] = -((a - 1) * 729 + (i - 1) * 81 + (j - 1) * 9 + d);
                    temp[2] = -((a - 1) * 729 + (i - 1) * 81 + (k - 1) * 9 + d);
                    addclausefull(newdouble, 2, temp);
                }
    }

//列约束
for (int a = 1; a <= 2; a++)
    for (int j = 1; j <= 9; j++) {
        for (int d = 1; d <= 9; d++) {
            for (int i = 1; i <= 9; i++)
                temp[i] = (a - 1) * 729 + (i - 1) * 81 + (j - 1) * 9 + d;
            addclausefull(newdouble, 9, temp);
        }
        for (int d = 1; d <= 9; d++)
            for (int i = 1; i <= 8; i++)
                for (int k = i + 1; k <= 9; k++) {
                    temp[1] = -((a - 1) * 729 + (i - 1) * 81 + (j - 1) * 9 + d);
```

```
                    temp[2] = -((a - 1) * 729 + (k - 1) * 81 + (j - 1) * 9 + d);
                    addclausefull(newdouble, 2, temp);
                }
            }

    //3*3 盒子约束
    for (int a = 1; a <= 2; a++)
        for (int horizontal = 1; horizontal <= 3; horizontal++)
            for (int vertical = 1; vertical <= 3; vertical++) {//3*3 的大盒子（循环其实也可以只用一个变量，依据%9 来确定位置）
                for (int d = 1; d <= 9; d++) {
                    int count = 0;
                    for (int i = (horizontal - 1) * 3 + 1; i <= horizontal * 3; i++)
                        for (int j = (vertical - 1) * 3 + 1; j <= vertical * 3; j++)
                            temp[++count] = (a - 1) * 729 + (i - 1) * 81 + (j - 1) * 9 + d;
                    addclausefull(newdouble, 9, temp);
                }
                for (int d = 1; d <= 9; d++)//9 个格子相互不能有重复数字
                    for (int i = (horizontal - 1) * 3 + 1; i <= horizontal * 3; i++)
                        for (int j = (vertical - 1) * 3 + 1; j <= vertical * 3; j++) {
                            int itemp = i;
                            while (itemp <= horizontal * 3) {
                                if (itemp == i)
                                    for (int k = j + 1; k <= vertical * 3; k++) {
                                        temp[1] = -((a - 1) * 729 + (i - 1) * 81 + (j - 1) * 9 + d);
                                        temp[2] = -((a - 1) * 729 + (i - 1) * 81 + (k - 1) * 9 + d);
                                        addclausefull(newdouble, 2, temp);
                                    } else
                                    for (int k = (vertical - 1) * 3 + 1; k <= vertical * 3; k++) {
```

```
                                        temp[1] = -((a - 1) * 729 + (i - 1) * 81 + (j
- 1) * 9 + d);
                                        temp[2] = -((a - 1) * 729 + (itemp - 1) *
81 + (k - 1) * 9 + d);
                                        addclausefull(newdouble, 2, temp);
                                    }
                                itemp++;
                            }
                        }
                }

    //中间位置的两个数独共享一个盒子
    for (int i = 7; i <= 9; i++)
        for (int j = 7; j <= 9; j++)
            for (int d = 1; d <= 9; d++) {
                temp[1] = -((i - 1) * 81 + (j - 1) * 9 + d);
                temp[2] = 729 + (i - 7) * 81 + (j - 7) * 9 + d;
                addclausefull(newdouble, 2, temp);
                temp[1] = (i - 1) * 81 + (j - 1) * 9 + d;
                temp[2] = -(729 + (i - 7) * 81 + (j - 7) * 9 + d);
                addclausefull(newdouble, 2, temp);
            }
}

int solvesudoku(formula &newdouble) { //将 DPLL 求解的结果转移到 board 数组内
    if (!DPLL(newdouble, 2))
        return 0;
    else {
        for (int i = 1; i <= newdouble.literalnum; i++)
            if (result[i]) {
                board[reverse(i) / 10] = reverse(i) % 10;
                fullboard[reverse(i) / 10] = reverse(i) % 10;
            }
```

```
        for (int i = 7; i <= 9; i++)
            for (int j = 7; j <= 9; j++)
                fullboard[200 + (i - 6) * 10 + (j - 6)] = fullboard[100 + i * 10 + j];//
同一个格子的另一种表示
        return 1;
    }
}

void printtwosudoku1(formula &newdouble) {//打印初始状态双数独棋盘
    for (int i = 0; i < 300; i++) {//初始化两个棋盘数组
        board[i] = 0;
        fullboard[i] = 0;
    }
    clause *findc = newdouble.head;
    int count = newdouble.singleclausenum;
    while (findc != NULL && count > 0) {
        if (findc->num == 1 && findc->head->word > 0) {
          board[reverse(findc->head->word) / 10] = reverse(findc->head->word) %
10;
            fullboard[reverse(findc->head->word) / 10] =
reverse(findc->head->word) % 10;
            count--;
        }
        findc = findc->next;
    }
    for (int i = 1; i <= 6; i++) {//第一个大数独
        printf("|");
        for (int j = 1; j <= 9; j++) {
            if (board[100 + 10 * i + j])
                printf("  %d", board[100 + 10 * i + j]);
            else
                printf("  _");
            if (j == 3 || j == 6 || j == 9)
```

```
                printf("    |");
            }
            printf("\n");
            if (i == 3)
                printf("————————————————————————————————————————\n");
            if (i == 6)

        printf("————————————————————————————————————————————————————————
—\n");
        }
        for (int i = 7; i <= 9; i++) { //重叠部分
            printf("|");
            for (int j = 1; j <= 9; j++) {
                if (board[100 + 10 * i + j])
                    printf("    %d", board[100 + 10 * i + j]);
                else
                    printf("    _");
                if (j == 3 || j == 6 || j == 9)
                    printf("    |");
            }

            for (int j = 4; j <= 9; j++) {
                if (board[200 + 10 * (i - 6) + j])
                    printf("    %d", board[200 + 10 * (i - 6) + j]);
                else
                    printf("    _");
                if (j == 3 || j == 6 || j == 9)
                    printf("    |");
            }
            printf("\n");
        }
        printf("
————————————————————————————————————————\n");
```

```
    for (int i = 4; i <= 9; i++) {//第二个大数独
        printf("                                    |");
        for (int j = 1; j <= 9; j++) {
            if (board[200 + 10 * i + j])
                printf("    %d", board[200 + 10 * i + j]);
            else
                printf("    _");
            if (j == 3 || j == 6 || j == 9)
                printf("    |");
        }
        printf("\n");
        if (i == 6)
            printf("
————————————————————————————————————————\n");
    }
}

void printtwosudoku2(formula &newdouble) {//打印完成状态双数独棋盘
    for (int i = 1; i <= 6; i++) {//第一个大数独
        printf("|");
        for (int j = 1; j <= 9; j++) {
            if (board[100 + 10 * i + j])
                printf("    %d", board[100 + 10 * i + j]);
            else
                printf("    _");
            if (j == 3 || j == 6 || j == 9)
                printf("    |");
        }
        printf("\n");
        if (i == 3)
            printf("————————————————————————————————————————\n");
        if (i == 6)
```

```
    printf("————————————————————————————————————————————————————————————————————————————————\n");
    }
    for (int i = 7; i <= 9; i++) { //重叠部分
        printf("|");
        for (int j = 1; j <= 9; j++) {
            if (board[100 + 10 * i + j])
                printf("  %d", board[100 + 10 * i + j]);
            else
                printf("  _");
            if (j == 3 || j == 6 || j == 9)
                printf("  |");
        }

        for (int j = 4; j <= 9; j++) {
            if (board[200 + 10 * (i - 6) + j])
                printf("  %d", board[200 + 10 * (i - 6) + j]);
            else
                printf("  _");
            if (j == 3 || j == 6 || j == 9)
                printf("  |");
        }
        printf("\n");
    }
    printf("
————————————————————————————————————————\n");
    for (int i = 4; i <= 9; i++) {//第二个大数独
        printf("                                  |");
        for (int j = 1; j <= 9; j++) {
            if (board[200 + 10 * i + j])
                printf("  %d", board[200 + 10 * i + j]);
            else
                printf("  _");
```

```
            if (j == 3 || j == 6 || j == 9)
                printf("    |");
        }
        printf("\n");
        if (i == 6)
            printf("
————————————————————————————————————\n");
    }
}

void digholes(formula &newdouble, int fullboard[300], int holecount) {
    int position;//挖洞位置，每次随机生成
    int trycount;//其它可能的解数量
    while (holecount > 0) {
        position = rand() % (299 - 111 + 1) + 111; //生成在 111 到 299 之间的一随机数
        if (!board[position])
            continue;
        while (board[position]) { //遍历确保挖去后解唯一
            if (!board[position])
                continue;
            trycount = 0;
            formula test;
            copyformula(test, newdouble);
            for (int i = 1; i <= 9; i++)
                if (i != fullboard[position])
                    temp[++trycount] = i;
            addclausefull(test, 8, temp);
            if (!DPLL(test, 2)) { //说明此处只能填这一个数，可以挖洞
                board[position] = 0;
                if (position == 177 || position == 178 || position == 179 || position == 187 || position == 188 || position == 189
                        || position == 197 || position == 198 || position == 199) //
```

```
重叠的位置
                    board[200 + ((position - 100) / 10 - 6) * 10 + position % 10 - 6]
= 0;
                if (position == 211 || position == 212 || position == 213 || position ==
221 || position == 222 || position == 223
                        || position == 231 || position == 232 || position == 233)
                    board[100 + ((position - 200) / 10 + 6) * 10 + position % 10 + 6]
= 0;
                break;
            } else //说明挖这个洞不合法
                while (1) {
                    position = rand() % (299 - 111 + 1) + 111; //生成在 111 到 299
之间的一随机数
                    if (board[position])
                        break;
                }
        }
        holecount--;
    }
}
```

附录二 程序使用说明

本程序具有完备的菜单选择提示功能，下面将简要说明使用流程。

初次进入程序后，请在 0~2 三个选项中选择一个，其中 0 代表退出程序，1 代表进行 SAT 问题求解，2 代表求解或生成双数独。

若选择 1，请继续在 0~4 五个选项中选择一个。若选择 0，则会回到上述初始菜单；若选择 1，则需输入 CNF 算例文件名称（若直接输入文件名，则要求 CNF 文件与程序位于同一目录下，否则需要使用绝对路径）。2~4 三项在读入 CNF 文件前均不可用。读入后，若选择 2，则会遍历输出全部子句；若选择 3，会对当前 CNF 文件进行 DPLL 求解并将答案输出到.res 文件中；若选择 4，则会清空之前读入的 CNF 文件，此时可以再次读入新文件。

若选择 2，请继续在 0~2 三个选项中选择一个。若选择 0，则会回到上述初

始菜单；若选择 1，则首先会展示一个初始的（未解出答案的）双数独棋盘，此时按任意键会开始对其的 DPLL 求解并将答案输出在屏幕上，再按任意键会显示求解好的双数独棋盘；若选择 2，则需要输入挖洞数量（一般在 30-110 个之间为宜），挖洞完成后会展示生成好的双数独棋盘，此时可根据提示开始填数。